

Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models

Assaf Eisenman¹, Kiran Kumar Matam¹, Steven Ingram¹, Dheevatsa Mudigere¹, Raghuraman Krishnamoorthi¹, Krishnakumar Nair¹, Misha Smelyanskiy¹, and Murali Annavaram^{1,2}

¹Facebook, Inc, ²USC

Abstract

Checkpoints play an important role in training long running machine learning (ML) models. Checkpoints take a snapshot of an ML model and store it in a non-volatile memory so that they can be used to recover from failures to ensure rapid training progress. In addition, they are used for online training to improve inference prediction accuracy with continuous learning. Given the large and ever-increasing model sizes, checkpoint frequency is often bottlenecked by the storage write bandwidth and capacity. When checkpoints are maintained on remote storage, as is the case with many industrial settings, they are also bottlenecked by network bandwidth. We present Check-N-Run, a scalable checkpointing system for training large ML models at Facebook. While Check-N-Run is applicable to long running ML jobs, we focus on checkpointing recommendation models which are currently the largest ML models with Terabytes of model size. Check-N-Run uses two primary techniques to address the size and bandwidth challenges. First, it applies differential checkpointing, which tracks and checkpoints the modified part of the model. Differential checkpointing is particularly valuable in the context of recommendation models where only a fraction of the model (stored as embedding tables) is updated on each iteration. Second, Check-N-Run leverages quantization techniques to significantly reduce the checkpoint size, without degrading training accuracy. These techniques allow Check-N-Run to reduce the required write bandwidth by 6-17× and the required capacity by 2.5-8× on real-world models at Facebook, and thereby significantly improve checkpoint capabilities while reducing the total cost of ownership.

1 Introduction

Deep learning has become extensively adopted in many production scale data center services. In particular, deep learning enabled recommendation systems power a wide variety of products and services. These include e-commerce marketplaces (e.g. Amazon, Alibaba) for recommending items to purchase [30, 33], social media platforms (e.g. Facebook, Twitter) for providing the most relevant content [14], entertainment services (e.g. Netflix, Youtube) for promoting new playlists [7, 12], and storage services (e.g. Google Drive) for enabling quick access to stored objects [4].

At Facebook’s datacenter fleet, for example, deep recommendation models consume more than 80% of the machine

learning inference cycles and more than 50% of the training cycles. Similar demands can be found at other companies [16].

Typically, the accuracy of deep learning algorithms increases as a function of the model size and number of features. For instance, the recommendation model size at Facebook grew more than 3× in under two years (see Figure 4). Recommendation models are particularly in need of massive model size to store sparse model features. Hence, they are orders of magnitude larger than even the largest DNNs, such as Transformer based models [32], and often occupy many terabytes of memory per model [38]. Because of their large size, these models also must be trained with massive datasets and run in a distributed fashion. Therefore, training recommendation models at production scale may take several days, even when training on highly optimized GPU clusters.

Given that the training runs span multiple GPU clusters over multiple days and weeks, there is an abundance of failures that a training run may encounter. These include network issues, hardware failures, system failures (e.g. out of memory), power outages, and code issues. Checkpointing is an important functionality to quickly recover from such failures for reducing the overall training time and ensure progress. Checkpoints are essentially snapshots of the running job state taken at regular intervals and stored in persistent storage. To recover from failure and resume training, the most recent checkpoint is loaded.

In addition to failure recovery, checkpoints are needed for moving training processes across different nodes or clusters. This shift may be required in cases such as server maintenance (e.g. critical security patches that could not be postponed), hardware failures, network issues, and resource optimization/re-allocation. Another important use-case of checkpoints is publishing snapshots of trained models in real time to improve inference accuracy (online training). For instance, an interim model can be used for prediction serving (obtained by checkpointing), while the model is still being trained over more recent dataset for keeping the inference model freshness. Checkpoints are also used for performing transfer learning, where an intermediate model state is used as a seed, which is then trained for a different goal [26].

Checkpoints must meet several key criteria:

(1) **Accuracy:** They must be accurate to avoid training accuracy degradation. In other words, when a training run is restarted from a checkpoint, there should be no perceivable

difference in the training accuracy or any other related metric. As has been stated in prior works on production scale recommendation models [38], even a tiny decrease of prediction accuracy would result in an unacceptable loss in user engagement and revenues. Hence, preserving accuracy is a constraint for checkpoint management in recommendation models.

(2) **Frequency:** Checkpoints need to be frequent for minimizing the re-training time (the gap between failure time and the most recent checkpoint timestamp) after resuming from a checkpoint. For instance, taking a checkpoint every 1000 batches of training data may lead to wasting time re-training those 1000 batches. Taking a checkpoint after 5000 batches leads to $5 \times$ more wasted work in the worst case. In the case of online training, the checkpoint frequency directly impacts how quickly the inference adapts in real time and its prediction accuracy.

(3) **Write Bandwidth:** Checkpoints at Facebook, as well as in other industrial settings, are written to remote storage to provide high availability (including replications) and scalable infrastructure. Writing multiple large checkpoints concurrently from different models that are being trained in parallel (e.g., thousands of checkpoints, each in the order of terabytes) to remote storage requires substantial network and storage bandwidths, which constitute a bottleneck and limit the checkpoint frequency. Hence, it is necessary to minimize the required bandwidth to enable frequent checkpoints.

(4) **Storage capacity:** Storing checkpoints at-scale requires hundreds of petabytes of storage capacity, with high-availability and short access times. Checkpoints at Facebook are typically stored for many days, thus the number of stored checkpoints at a given time is reflected by the number of training jobs in that time period. While the last checkpoint per run is saved by default, it is often useful to keep several recent checkpoints (e.g. for debugging and transfer learning). As models keep getting larger and more complex, resulting in an ever increasing storage capacity demand, it is necessary to reduce the corresponding checkpoint size to minimize the required storage capacity for accommodating all checkpoints.

Unfortunately, standard compression algorithms such as Zstandard [6] are not useful enough for deep recommendation workloads. In our experience, we were able to reduce the checkpoint size and the associated write-bandwidth and storage capacity by at most 7% using Zstandard compression.

Based on the above challenges, we present Check-N-Run, a high-performance scalable checkpointing system, particularly tailored for recommendation systems. Check-N-Run’s main goal is to significantly reduce the required write-bandwidth and storage capacity, without degrading accuracy. Our goal is to work within the accuracy degradation constraint set by business needs ($< 0.01\%$).

Recently, CheckFreq has demonstrated the benefits of checkpointing for deep neural networks(DNNs) [19]. CheckFreq proposed *adaptive rate tuning* to dynamically determine when to initiate a checkpoint, and a *two-phase* strategy to

enable checkpoint storage and training to move concurrently. However, recommendation models provide unique opportunities to tackle checkpointing challenges that are not afforded in traditional DNNs. First, recommendation models update only part of the state after every batch. Hence, it is possible to explore checkpointing strategies that can incrementally store the checkpoint. Second, recommendation model sizes can exceed Terabytes, which stress even planetary scale storage systems. Check-N-Run builds on several techniques:

(1) **Differential checkpointing:** Check-N-Run utilizes differential checkpointing for reducing the checkpoint write bandwidth. This is a technique that is particularly well suited for recommendation models where only a small fraction of the model parameters are updated after each iteration. This is a unique property of recommendation models. In traditional DNNs the entire model is updated after each iteration since gradients are computed for all the model parameters. Recommendation models, on the other hand, access and update only a small fraction of the model during each iteration. Differential checkpoints leverage this observation by tracking and storing the modified parts of the models.

(2) **Quantization:** Check-N-Run leverages quantization techniques to significantly reduce the size of checkpoints. This optimization reduces the required write bandwidth to remote storage, and the storage capacity. While quantization of model parameters during training may have a negative impact on accuracy, checkpointing has the advantage that quantization is only used to store the checkpoint, while full precision is used for training. The only time checkpoint quantization may impact training accuracy is when the quantized checkpoint is restored and de-quantized to resume training. Check-N-Run leverages this insight to maintain training accuracy within our strict bounds.

(3) **Decoupling:** To minimize the run time overhead and training stalls, Check-N-Run creates distributed snapshots of the model in multiple CPU host memories. That way, training on the GPUs can continue while Check-N-Run is optimizing and storing the checkpoints in separate processes running on the CPU in the background. Check-N-Run enables the frequent checkpointing of hundreds of complex production training jobs running in parallel over thousands of GPUs, each job training a very large model (in the order of terabytes). This decoupling approach is also proposed in CheckFreq which separates *snapshot* process from the *persist* storage process [19]. Our implementation of decoupled checkpointing leads to less than 0.4% of time when the trainer processes must pause to take a snapshot. Hence, the impact of taking a checkpoint on the *training speed is negligible*.

The contributions in this paper include:

- (1) To our knowledge, Check-N-Run is the first published checkpointing system that uses quantization and differential views for recommendation systems at-scale, demonstrated on real-world workloads.
- (2) We design and evaluate a wide range of checkpoint quanti-

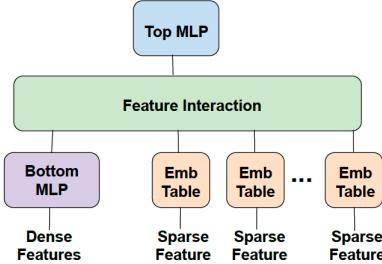


Figure 1: A typical recommendation model. It consists of large embedding tables for mapping the sparse features to vectors, and MLPs for processing the dense features (bottom MLP). These feature interactions are combined in the top MLP. The interaction op combines the dense and sparse features, in order to train with them together.

zation approaches to significantly reduce the checkpoint size by $4\text{--}13\times$, without degrading the training accuracy.

- (3) We introduce differential checkpoints, which store the modified part of the model, rather than storing the entire model. Differential checkpoints reduce the average write bandwidth by more than 50%, with no impact on accuracy.
- (4) Finally, we demonstrate a heterogeneous checkpointing mechanism that combines differential checkpointing with quantization. Check-N-Run provides $6\text{--}17\times$ improvement in the required checkpointing write bandwidth, and $2.5\times\text{--}8\times$ less capacity, without sacrificing accuracy and run time.

2 Background

2.1 Recommendation Models

Recommendation models are a variant of deep learning models that are used to provide recommendations to users based on their past interactions with a digital service. Recommendation systems are often used in commercial settings and dominate the datacenter capacity for AI training and inference [22]. Broadly speaking, recommendation models use a combination of a fully connected multi-layer perceptron (MLP) to capture the dense features, and a set of sparse features that capture categorical data such as a user’s past activity and main characteristics of a post. Figure 1 depicts a typical recommendation model used in this study.

Sparse features are captured through embedding tables [10], which map each category to a dense representation in an abstract space. Each embedding table may contain many millions of vectors, with different vector dimensions (e.g. 64), where each element is a 32-bit floating-point number during training. Embedding tables constitute the majority of the model footprint, and account for $>99\%$ of its size. A training sample includes a set of vector indices per embedding table, which is used to extract the corresponding multi-hot encoded vectors stored in those indices. Once the embedding vectors are extracted, they are trained with a deep neural network.

The size of the sparse layer prevents the use of pure data

parallel training, since it would require replicating the large embedding tables on every device. The large footprint of the sparse layer requires the distribution of the embedding tables across multiple devices, emulating model parallelism. MLP parameters, on the other hand, have a relatively small memory footprint, but they consume a lot of compute. Hence, data-parallelism is an effective way to enable concurrent processing in the MLPs, by running separate samples on different devices and accumulating the updates. Our training system thus uses a combination of model parallelism for the sparse layer, and data parallelism for the MLPs. This hybrid approach mitigates the memory bottleneck of accessing the embedding tables by distributing these tables across multiple GPUs, while parallelizing the forward and backward propagation over the MLPs.

2.2 High Performance Training at Facebook

Given the prominence of recommendation models in today’s social media platforms, these models are trained on dedicated clusters [23, 38]. At Facebook, over 50% of the ML training cycles are dedicated solely to recommendation models. Figure 2 illustrates the training pipeline for deep learning recommendation models. Broadly speaking, it consists of 3 stages, located at separate clusters: dataset reader cluster, training cluster, and remote checkpoint storage.

To support high-performance training, our training system relies on clusters of GPUs attached to host CPUs as shown in Figure 2 (*training cluster*). The GPUs accelerate the training tensor operations and accommodate the model parameters, while CPUs run other tasks, such as data ingestion and checkpoint handling. Each training cluster contains 16 nodes, each with 8 GPUs attached to multi-core CPU. Hence, training a model on an entire cluster would partition the embedding tables and the training batches over 128 GPUs, in addition to replicating the MLPs over these GPUs.

In cases where GPU memory is not sufficient for accommodating the models, our training system leverages hierarchical memory: the model parameters are stored in DRAM, while GPU memory serves as a cache.

The model parameters are updated *synchronously* [3], ensuring the updated parameters across the devices are consistent before each training iteration. This is needed for enabling scalable training and avoiding accuracy degradation when training in high throughput. Fully synchronized training avoids regression in the model quality with increased scale and decouples model quality from training throughput. We employ a decentralized model synchronization approach in which each node performs the computations on its local part of the model. For the data-parallel MLPs, an “AllReduce” communication is done in the backward pass to accumulate the gradients computed on the multiple GPUs (each with a different sub-batch of data). For the model-parallel sparse layer, an “AlltoAll” communication [23] occurs both in the forward

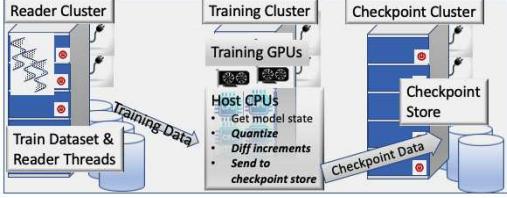


Figure 2: An Overview of Training and Checkpoint Systems

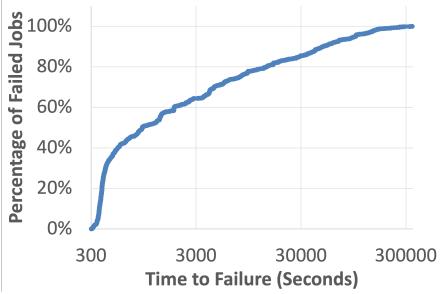


Figure 3: Training job failure CDF in our cluster. Jobs that fail within 5 minutes are removed since they are usually simple user setup errors.

pass (to communicate the looked-up embedding vectors), and in the backward pass (to communicate the embedding vector updates). Checkpoint write process is done in the background (using dedicated CPU processes in the trainer nodes), while the training process continues in GPU.

Since the dataset used for training (i.e., training samples) is enormous, and training has to be done at high-throughput (e.g. 500K training queries per second called QPS), it is important to make sure that reading training data will not become a bottleneck. As such, the training system deploys a separate distributed reader tier (shown as *Reader Cluster* in Figure 2), which enables reading resources and training resources to scale independently. Each training cluster uses hundreds of reader nodes residing in a separate cluster, in charge of saturating the trainer with training samples.

Checkpoints of the training job state (consisting of both the reader and trainer states) are stored at a separate, remote storage (shown as *Checkpoint Cluster* in Figure 2).

Training jobs are submitted to this infrastructure through an internally developed job scheduling interface. Schedulers like Bistro [11] and PBS [15] handle job and user priorities, and manage the job queue. The scheduler assigns jobs based on the job configuration and cluster availability. It continuously monitors both the job progress as well as system health status.

3 Motivation

3.1 Training Failures

We analyzed the training job failures on a training system consisting of 21 training clusters, over a one month period. Figure 3 presents the time-to-failure statistics. The X-axis shows the total execution time that was completed by a job before it failed, and the Y-axis shows the percentage of failed jobs which failed before that time. The data shows that longest

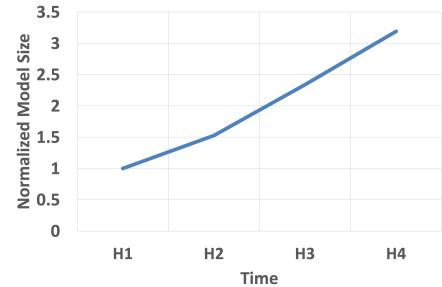


Figure 4: The normalized model size over the past 2 years

10% of the failed jobs ran for at least 13.5 hours before they fail, and the top 1% of the failed jobs fail after executing for not less than 53.9 hours. Note that many of these jobs require 128 GPUs spanning many nodes, that are expensive to maintain and run. These training jobs interact with multiple systems for training. For instance, the training process accesses training samples provided by a separate reader cluster. As such, any one failure in these inter-connected systems will hobble the entire training progress. This data shows the criticality of efficient checkpointing to ensure training progress. Otherwise, long running training jobs may never complete their task. This data motivated the need for Check-N-Run.

As the model sizes are growing continuously, training is getting distributed even more widely across the datacenters. Hence, the failure rates are expected to continue to grow significantly. Thus checkpointing of large model training is a critical problem for any production model.

3.2 Model Size

Recommendation model sizes are often massive due to their large sparse features (represented as embedding tables). Typically, the accuracy of these models increases as a function of the model size. Figure 4 shows our model size increase over the past 2 years (exact model size is confidential). As can be seen, it increased by over 3×. Given the large and ever-increasing model sizes, checkpoints are often bottlenecked by write-bandwidth and storage capacity.

3.3 Model Updates

Another set of motivation data shows the sparsity of model updates over time. We analyze one of the largest recommendation models at Facebook and observe that due to large model sizes and their high sparsity, only a fraction of the embedding vectors is modified in a given training interval. Figure 5 shows the percentage of the model that is modified, as a function of the number of training records used to train, starting from three different initial states. The curve starting at the origin shows what fraction of the model size is updated starting from the first training record and ending at about 11 billion training records. As can be seen, even after 11 billion training records, the fraction of the model that is accessed grows slowly and

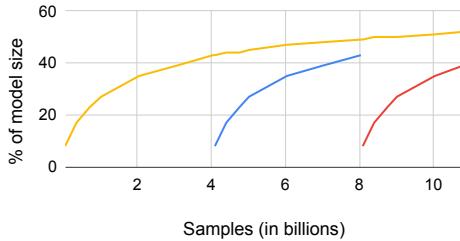


Figure 5: The fraction of model size modified w.r.t. the number of training samples, measured from 3 different starting points

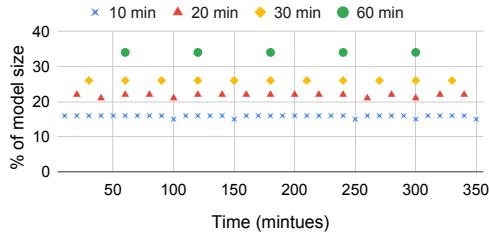


Figure 6: The fraction of model size that is modified during different time intervals

reaches only 52%. Furthermore, the fraction of the model updated during a training interval is expected to continue to shrink as model sizes keep increasing, which is the general industry trend.

The second curve in Figure 5 shows how the fraction of the updated model grows if we only observe updates starting at the 4 billionth training record. The third curve shows the same data starting at about the 8 billionth training record. It is interesting to note that no matter when we start observing the model size growth, the fraction of the modified model size follows a similar slope. This fact is made more clear in Figure 6, which plots the fraction of model size that is modified during a given time interval. For a given interval length, the fraction of model size that is modified remains almost the same in all intervals (e.g., in each 30 minute intervals, about 26% of the model is modified). The above data indicates that at each iteration only a tiny fraction of the model is updated.

4 Check-N-Run Design Overview

Check-N-Run is a distributed checkpointing system for training systems at scale, implemented in our PyTorch training framework. Check-N-Run generates accurate checkpoints of the training system state and ensures there is no accuracy degradation due to creating or loading from a checkpoint. Since training accuracy is a main concern, we are not interested in exploring choices that come at the expense of an unacceptable training accuracy loss, even as small as 0.01%. In this section, we provide an overall overview, while in section 5 we discuss the checkpoint optimization details. Figure 7 illustrates Check-N-Run’s overall design, showing what functionality is implemented in each of the reader, trainer and checkpoint storage tiers. Check-N-Run is implemented pri-

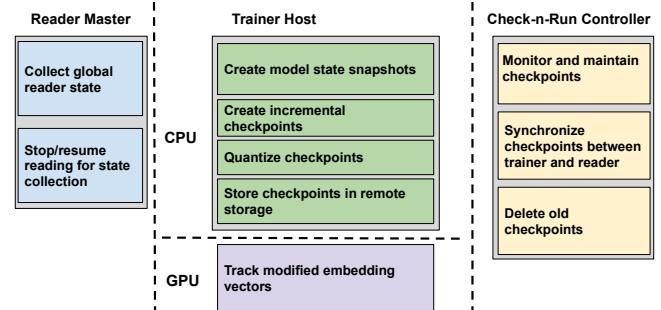


Figure 7: Check-N-Run design components

marily on the host CPU of the training cluster, while its tracking mechanism (described in 5.1.1) is implemented in GPU. It has additional coordination threads running on the reader master (in the reader cluster) and a lightweight Check-N-Run controller that may reside in a dedicated host. Checkpoints are written to remote object storage to provide high availability (including replications) and storage scalability.

4.1 What to Checkpoint?

The trainer state consists of all the model layers (including the sparse and dense features), the optimizer state, and the relevant metrics. Since the MLPs are replicated and maintained with a consistent view during training, it is enough to read them from a single GPU for checkpointing. The embedding tables, however, are distributed across GPUs and hence each GPU must provide a snapshot of the embedding tables that are stored in its local memory.

When a training job resumes from a checkpoint, the run should still train the same training dataset as the original run. Hence, the checkpoint must also include the reader state. This is important, for example, to avoid training the same sample twice. The reader reads the dataset in the granularity of splits (each split represents successive rows of the dataset). Its state includes the set of splits that are pending, and the set of splits that have been partially read (including their cursor position). Note that checkpoints that are intended solely for alternate use-cases such as online training (frequently updating an already trained model running in inference) and transfer learning, do not require the reader state.

Avoiding the trainer-reader state gap: In a production scale training system, checkpointing has unique challenges. As described earlier, a separate set of distributed readers is in charge of feeding the trainers with batches in sufficient throughput. Since readers and trainers work in a distributed fashion in our training system (and reside in separate clusters), many training records are in-flight and reside in different queues. These are batches that have been read by the reader, but have not been consumed by the trainer yet. They constitute a gap between the reader state and the trainer state, which may affect accuracy when loading from a checkpoint. After resuming from a checkpoint, the reader may not know which of

the training samples have been processed. To avoid this gap, Check-N-Run’s controller communicates to a *coordination thread* running on the reader master how many batches to read until the next checkpoint. The reader makes sure to read this exact number of batches. For example, if the checkpointing interval is 1000 batches, the reader will provide exactly 1000 batches to the trainer and then stop reading. When trainer finishes processing the 1000th batch and a checkpoint is triggered, there will be no in-flight batches. That way, there is essentially no gap between the reader state and the trainer state. After reader state has been collected, Check-N-Run signals the reader to resume reading the number of batches until the next checkpoint.

4.2 Decoupled Checkpointing

Checkpointing requires the model parameters to be atomically copied for further processing and storage. Note that this atomicity is important for consistency. Otherwise, training processes may update the model during the copying time window, causing substantial consistency challenges and potential accuracy degradation when loading checkpoints. Check-N-Run achieves atomicity by stalling training at the start of a checkpoint and transferring the model state from GPU memory to host CPU memory. Training is stalled only when creating a copy of the model parameters in-memory. As soon as the model snapshot is ready, dedicated CPU processes are in charge of creating, optimizing, and storing checkpoints in the background, while training continues on the GPUs. All training nodes concurrently create a unique snapshot of their own local part of the model. For instance, if the embedding tables are distributed across multiple nodes, each node snapshots its own embedding tables and transfers that information to the host CPU.

Using this approach to create a snapshot scales well with larger models and more nodes, as utilizing additional nodes does not increase the checkpoint performance overhead. For instance, creating a snapshot (in CPU DRAM) of a typical model residing in the GPU memory and partitioned across 16 nodes, each with 8 GPUs (total of 128 GPUs), would stall training in our system for less than 7 seconds. When checkpointing every 30 minutes (our default), stall time would be a negligible fraction (< 0.4%).

4.3 Checkpointing Frequency

The checkpointing frequency is bounded by the available write bandwidth to remote storage. Since Check-N-Run leverages remote storage, it is also limited by available network bandwidth. With larger and ever increasing model sizes, as well as the growing number (e.g. hundreds) of training clusters that concurrently train and checkpoint separate training jobs, these resources constitute a bottleneck. In Check-N-Run, two consecutive checkpoints cannot overlap, and writing of

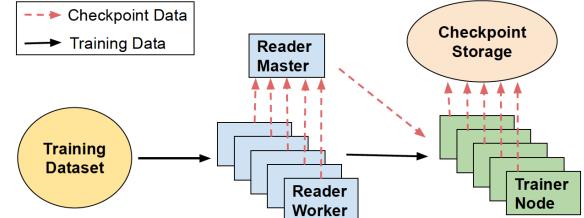


Figure 8: High-level data flow during training

the current checkpoint must be completed or cancelled before a new checkpoint can be created. That way, the current checkpoint can utilize all available resources to minimize the write latency (i.e., the time it takes a checkpoint to become valid and ready to use). Based on our model size and system resource considerations, we initiate a new checkpoint every 30 minutes by default. In section 5 we describe the optimizations leveraged by Check-N-Run to significantly reduce the required resources, providing a scalable solution to enable high frequency checkpointing and reduce the associated total cost of ownership (TCO).

4.4 Check-N-Run Workflow

We define the *checkpoint interval* as the number of trained batches between two consecutive checkpoint. The checkpoint operation is triggered at the end of each checkpoint interval (a configurable number of batches), after the backpropagation stage of the last batch in that interval. Since our training system is fully synchronous, all GPUs will reach their last batch in the checkpoint interval and wait until the next batch is started. The checkpointing process consists of 3 main stages: (1) Create an in-memory snapshot of the training state (2) Build an optimized checkpoint (3) Write the checkpoint to storage.

Figure 8 depicts the high-level data flow between the reader, trainer, and remote checkpoint storage during training.

At the beginning of the training run, Check-N-Run’s controller communicates to the coordination thread on the reader master node in the reader tier, to inform what is the checkpoint interval, i.e. how many batches to read until the next checkpoint. The reader master then initiates several reader worker threads which start reading data from the training dataset to provide the trainer nodes. When a checkpoint is triggered, Check-N-Run collects the reader state at this point, which specifies what parts of the training dataset have been read so far. At the same time, all trainer nodes are stalled to concurrently create a snapshot of their local state, by copying the model state from each of their GPUs into host CPU DRAM. As soon as all snapshots are ready, training continues. This decoupling mechanism essentially minimizes the checkpointing process from bottlenecking the trainer.

In step 2, Check-N-Run leverages several techniques to reduce the required checkpoint capacity and write bandwidth, as described in section 5. These techniques are concurrently

applied by each trainer node and run in dedicated CPU processes that are resident on the host CPUs in the trainer tier, outside of the GPU critical path. Only the tracking mechanism described in 5.1.1 is implemented in GPU.

In step 3, the checkpoint is moved to remote checkpointing storage. Note that the optimization process in step 2 works on chunks of embedding vectors at a time. Hence these chunks of quantized and differential checkpoints can be stored in a pipelined manner, enabling concurrent optimization and the checkpoint storing process. When all nodes finish storing their part of the checkpoint successfully, Check-N-Run’s controller will declare a new valid checkpoint. At that stage, an older checkpoint may be deleted by the controller (based on the system configuration). Multiple checkpoints can be stored depending on the needs and use cases.

5 Checkpoint Optimizations

5.1 Differential Checkpointing

One-Shot Differential Checkpoint: Motivated by the insight presented in Section 3 regarding the fraction of the model size that is modified after each iteration, we introduce differential checkpoints. Differential checkpointing starts with a single checkpoint taken as a full baseline checkpoint, including all the embedding vectors. From this point, the system starts tracking all modified vectors to create a differential view of the embedding vectors that would have to be included in the next checkpoint. Each differential checkpoint would then store only the vectors that were modified since the baseline checkpoint. To resume from a checkpoint, both the baseline checkpoint and the most recent differential checkpoint have to be read. We refer to this method as *One-shot baseline*.

Consecutive Incremental Checkpoint: We also explored an alternative way, which we denote as *consecutive incremental checkpoint*. This approach stores the vectors that were modified only during the last checkpoint interval, rather than storing the vectors from a baseline checkpoint. This method reduces the current checkpoint size, since only those modified vectors since the last interval are stored. But this approach would require keeping all previous incremental checkpoints for reconstructing the model when resuming from a checkpoint. Note that in our remote object storage, merging consecutive incremental checkpoints would require moving all the data back to the CPU host, which costs substantial bandwidth. Keeping all the incremental checkpoints leads to higher storage capacity since a vector that is modified during multiple intervals will have multiple copies stored. However, incremental checkpoints are useful for use cases such as online training, where checkpoints are directly applied to an already-trained model in inference to improve its freshness and accuracy.

Intermittent Differential Checkpoint: One challenge with the above methods is that the checkpoint size gradually increases. As training progresses, the number of modified model

parameters over a baseline will increase. One way to reduce this growth is to checkpoint a full model intermittently, so that the differential view size can be reduced. We exploit the observation from Figure 5 that the modified model size grows similarly from three different starting points.

We use a simple history based predictor to decide when to take a full checkpoint. At the end of each checkpoint interval, it estimates the expected cumulative size of future checkpoints if another differential checkpoint is taken, compared with the total expected size if a full checkpoint is taken (which will then reduce the future checkpoint sizes). Based on this comparison, the system decides whether to take a full checkpoint or stay with a differential checkpoint. The algorithm for this selection is as follows:

Let S_1, S_2, \dots, S_i be the sizes of the past i differential checkpoints, which follow a full baseline checkpoint with a size S_0 . S is expressed as a fraction of the full baseline checkpoint, such that $S_0 = 1$. Then, at the $(i+1)^{th}$ interval, Check-N-Run faces two options: (1) create a full baseline checkpoint, or (2) create another differential checkpoint. If a full baseline checkpoint is created, we estimate the future cumulative checkpoint size F_c of the next $i+1$ intervals to be similar to the past $i+1$ intervals. That is, $F_c = 1 + S_1 + S_2, \dots, +S_i$. Alternatively, if a differential checkpoint is created, the total checkpoint size of the next $i+1$ intervals is larger than, or at best equal to $I_c = (i+1) * S_i$. This relation holds, because starting at interval $i+1$ differential checkpoint size will be at least S_i . Thus, at the $(i+1)^{th}$ interval, we do a full checkpoint if $F_c \leq I_c$, else we do a differential checkpoint. We term this approach as *intermittent differential checkpoint*. This approach can be improved with more accurate prediction models, which are part of future work.

5.1.1 Efficient Tracking

Check-N-Run is intended for high-performance training, hence it aims to minimize the overhead of tracking which embedding vectors are modified. Since the embedding tables are partitioned across the GPUs, each GPU separately tracks the accesses to its local embedding tables. For the sake of simplicity, the training records are tracked during the forward pass, as most of the embedding vectors accessed in the forward pass are also modified during the backward pass. During tracking, each GPU updates a bit-vector associated with its local embedding vectors. This bit-vector is used as a mask to determine which embedding vectors are modified during the training process, and should eventually be included in the next differential checkpoint. Note that the bit-vector memory footprint is low (typically less than 0.05%, on the order of several MBs per GPU).

We utilize idle GPU cycles to reduce tracking overhead, by scheduling the tracking functionality during the “AlltoAll” communication phase (described in section 2.2). Using this implementation, the tracking overhead is reduced to $\approx 1\%$ of

the iteration training time.

5.2 Checkpoint Quantization

The second technique that Check-N-Run uses is quantization of checkpoints. While quantization has been adopted in some cases for reducing model size during inference [18, 37, 40], or to reduce communication costs of parameter aggregation [36], training is typically done in single-precision floating-point format (FP32) to maximize training outcomes and model accuracy. Check-N-Run leverages quantization techniques to significantly reduce the checkpoint size during training, without sacrificing training accuracy.

Quantization in Check-N-Run is decoupled from the training process and is done in background CPU processes after a model snapshot has been created. Hence, it does not affect training throughput. Since quantization is applied to a chunk of rows, the quantized checkpoint store operation does not have to wait until the entire checkpoint is quantized and can store the quantized rows eagerly as needed.

The quantization of embedding tables is usually applied in the granularity of an entire embedding vector. We aim to minimize the error between the original vector $X \in \mathbb{R}^n$ and the quantized vector $Q \in \mathbb{Z}^n$, by minimizing $\|X - Q\|_2$. We define the *mean ℓ_2 error* of an entire quantized checkpoint as: $\frac{1}{m} \sum_{i=0}^m \|X_i - Q_i\|_2$, where m is the total number of embedding vectors in the checkpoint. The *mean ℓ_2 error* metric is a good proxy for accuracy loss because the model accuracy is dependent on the values of the embedding tables. This metric captures the distance between the original value of an embedding entry without quantization and the new value produced due to quantization. We observed that this difference provides the first order impact on the accuracy loss and use it to compare different quantization methods. In section 6, we demonstrate how training accuracy is impacted by Check-N-Run’s quantization schemes.

In this work we explored 3 quantization methods, *Uniform Quantization*, *Non-Uniform Quantization* and *Adaptive Quantization*, to empirically evaluate which approach provides the lowest *mean ℓ_2 error*. Let x be the value of an element in an embedding vector $X \in \mathbb{R}^n$, clipped to the range $[x_{min}, x_{max}]$. N-bits quantization maps x to an integer in the range $[0, 2^N - 1]$, where each integer corresponds to a quantized value. If the quantized values are a set of discrete, evenly-spaced grid points, the method is called *uniform quantization*. Otherwise, it is called *non-uniform quantization*. We describe these approaches in detail next.

Approach 1: Symmetric-vs-Asymmetric Uniform Quantization: Uniform quantization maps the embedding table values into integers in the range $[0, 2^n - 1]$. It relies on two parameters: *scale* and *zero_point*. *Scale* specifies the quantization step size, and is defined as $scale = \frac{x_{max} - x_{min}}{2^n - 1}$, while *zero_point* is defined as x_{min} . The quantization proceeds as follows: $x_q = round\left(\frac{x - zero_point}{scale}\right)$. The de-quantization op-

eration is: $x = scale * x_q + zero_point$. We denote uniform quantization as $F_Q(x, x_{min}, x_{max})$.

In *symmetric* quantization, x_{max} is set by the maximum absolute value in X , and $x_{min} = -x_{max}$. This is a very simple approach to quantize. An improved approach is to pick x_{min} and x_{max} to use the minimum and maximum element values that are actually present in an embedding vector. We refer to this method as *asymmetric* quantization. Asymmetric quantization, however, has the small additional overhead of storing of both x_{min}, x_{max} values for de-quantization process.

Figure 9 shows the mean ℓ_2 error of symmetric (first bar) and asymmetric quantization (second bar) for different bit-widths used in quantization. Since the elements of the embedding vectors are not symmetrically distributed, asymmetric quantization consistently outperforms symmetric quantization. Note that we generated this result from one representative checkpoint created after training a production dataset for about 18 hours.

Approach 2: Non-uniform Quantization using K-means

We explored non-uniform quantization where embedding vectors are not all mapped into equally spaced buckets. This approach is useful when the elements in a typical embedding vector are not necessarily uniformly distributed.

We leverage the unsupervised K-means clustering algorithm for clustering elements in the embedding vector $X \in \mathbb{R}^n$ into groups. For N-bits *k-means quantization*, the n elements in X are partitioned into 2^N clusters. Let C_i be the cluster i with a corresponding centroid c_i . K-means quantization maps the element $x \in C_i$ to the integer $x_q = i$. In addition, it keeps a codebook entry, such that $codebook[i] = c_i$. The de-quantization operation in that case is: $x = codebook[x_q]$

Figure 9 shows that the third bar in each group, labeled *k-means per vector*, provides lower mean ℓ_2 error compared with asymmetric quantization, when running k-means with 15 iterations. Note that K-means performs slightly worse than asymmetric for a bit-width of 4, due to cluster initialization randomness. While mean ℓ_2 error metric is marginally better, the run time of K-means clustering algorithm was orders of magnitude slower than uniform quantization. For instance, performing K-means clustering using off-the-shelf clustering packages on just one checkpoint of our production training model took more than 48 hours. This is not surprising since prior works have acknowledged the challenge of K-means clustering on large datasets and advocated for sampling a small fraction of the dataset to reduce their overheads [21]. We have explored different approximate clustering strategies but approximations yielded substantial mean ℓ_2 error. Hence, when taking into account any incremental benefits of clustering against the cost of running the clustering algorithm for checkpointing, we conclude that k-means is not feasible in Check-N-Run.

Approach 3: Adaptive Asymmetric Quantization: We observe that the naive way of setting x_{min} and x_{max} in asymmetric quantization may not be optimal in some cases. For example,

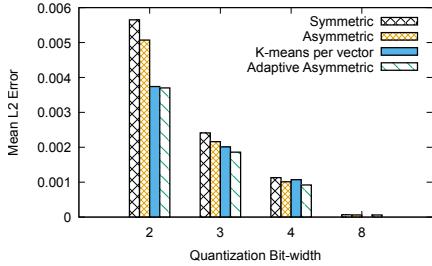


Figure 9: Mean ℓ_2 error of a quantized checkpoint for different quantization approaches

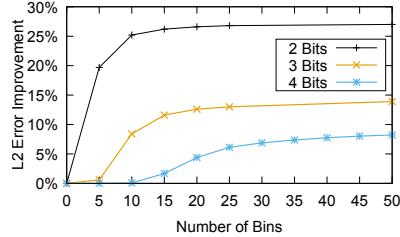


Figure 10: Mean ℓ_2 error improvement of adaptive asymmetric quantization over naive asymmetric quantization for different bit-widths, as a function of bins

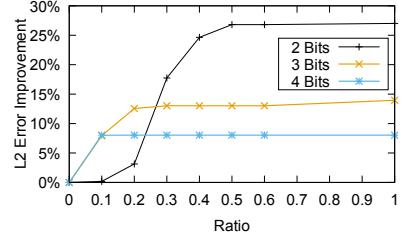


Figure 11: Mean ℓ_2 error improvement for different bit-widths, as a function of the number range ratio (after selecting optimal number of bins)

if a vector contains an element with a relatively high absolute value compared with the other elements, *scale* may be too high.

A brute force approach for selecting more optimal x_{min} and x_{max} values for each embedding vector would iterate over many possible values, and in each iteration perform a quantization for the sole purpose of measuring ℓ_2 error. Based on that, it would choose the x_{min} and x_{max} values that provided the lowest ℓ_2 error. Unfortunately, since this has to be done per embedding vector, it is not feasible in terms of run time when quantizing large models.

To address this issue, Check-N-Run leverages a greedy search algorithm [13] to select the x_{min} and x_{max} values per embedding vector. We define *step_size* as the the original range of the vector divided by a configurable number of bins: $step_size = \frac{X_{max} - X_{min}}{num_bins}$. At each iteration, two quantizations are performed for the sole purpose of comparing their ℓ_2 error: $F_Q(x, x_{min} + step_size, x_{max})$ and $F_Q(x, x_{min}, x_{max} - step_size)$. Based on the update that provided a lower ℓ_2 error, either x_{min} or x_{max} are set to $x_{min} + step_size$ or $x_{max} - step_size$, respectively. Finally, when all iterations are done, the optimal x_{min} and x_{max} are chosen from the iteration with the lowest ℓ_2 error.

The greedy algorithm contains a configurable parameter, *num_bins*, which determines its step size. In addition, we add a *ratio* parameter, which determines the fraction of the original $range = X_{max} - X_{min}$ to iterate over. In other words, the greedy algorithm would iterate as long as $x_{max} - x_{min} < ratio * range$. For example, when *ratio* is set to 1, the algorithm would iterate over the entire range. If *ratio* is 0.6, the algorithm would stop once it covered 60% of the original range. While decreasing the number of bins and ratio both reduce run time, it may also result higher ℓ_2 error. Figure 10 demonstrates the mean ℓ_2 error improvement of adaptive asymmetric quantization over naive asymmetric quantization for different bit-widths, as a function of the number of bins.

Figure 11 depicts the mean ℓ_2 error improvement for various range ratios, based on the optimal number of bins from figure 10 (25 bins for bit-widths of 2 bits and 3 bits, and 45 bins for 4 bits). As can be seen, lower bit-width quantizations

are more sensitive to the ratio parameter (and also gain higher improvement by the adaptive asymmetric).

Parameter selection: Check-N-Run automatically sets the greedy algorithm parameters by performing a light-weight checkpoint profiling. It uses the insight that mean ℓ_2 error can be estimated efficiently without having to quantize the entire checkpoint. It uniformly samples a small fraction of the checkpoint (0.001% by default), then quantizes the sampled checkpoint with different parameter values and calculates the mean ℓ_2 errors. With this method, Check-N-Run is able to identify the optimal parameter by automatically choosing the parameter in which the mean ℓ_2 error improvement tapers off. In our experiments, the sampled checkpoint provided identical parameter selection compared with the full checkpoint.

In section 6.1, we evaluate the quantization latency as a function of *num_bins* and *ratio*.

Summary of various approaches: Based on these empirical data, Check-N-Run utilizes adaptive asymmetric quantization for bit-width of 4 bits or less. As shown in figure 9, adaptive asymmetric quantization perform similarly to k-means quantization. For 8-bit quantization, naive asymmetric quantization is sufficient. The quantization bit-width itself is determined dynamically by the expected number of times a training job would resume from a checkpoint, as we elaborate in section 6.

6 Experimental Evaluation

In this section, we evaluate the performance implications of Check-N-Run, its training accuracy implications, and the achieved write bandwidth and storage capacity reduction. We implemented Check-N-Run in our PyTorch training framework and evaluate it in our high-performance training clusters, under production scale models and training datasets.

We use clusters of NVidia HGX-like nodes [25] for training, with some customization such as increased host memory of up to 1.5 TB of DRAM per node, up to 56 cores per node, and alternate scale-out fabric such as NVSwitch and NVLinks (connecting up to 16 nodes). Each GPU is able to communicate directly with GPUs on a different node through a dedicated RoCE NIC, without involving a host CPU. In addition, there is a front-end NIC connected to each CPU. Checkpoints

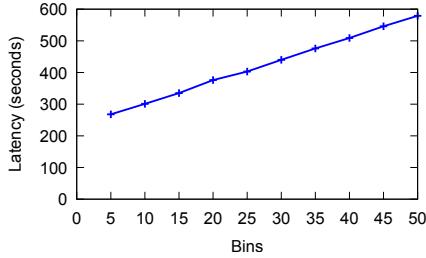


Figure 12: Total checkpoint quantization latency when using adaptive asymmetric quantization, as a function of the number of bins used by the greedy algorithm (ratio=1.0)

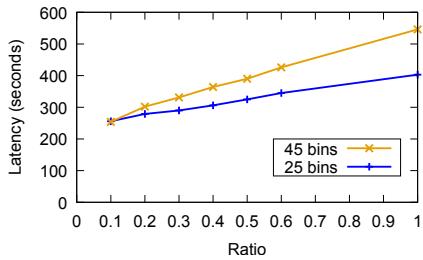


Figure 13: Total checkpoint quantization latency when using adaptive asymmetric quantization, as a function of the ratio used by the greedy algorithm with 25 and 45 bins

are written to remote storage through the regular front-end network, without interfering with inter-GPU communication.

6.1 Performance

Checkpoint overhead on training: Check-N-Run decouples checkpointing from training by creating an in-memory snapshot of the model state before checkpointing. This enables training to continue while checkpoints are created, optimized, and stored in the background. Check-N-Run creates snapshots by copying the model state from GPU's HBM to pinned CPU memory. We measured this operation to take up to 7 seconds in our setting, during which training is stalled. When checkpoint intervals are 30 minutes, the default setting, that overhead translates to less than 0.4% reduction in training throughput.

Tracking the modified embedding vectors in each training iteration requires updating a local bit vector, which is used to mark the modified embedding vectors in the current checkpoint interval. As described in 5.1.1, our efficient implementation uses idle GPU cycles to hide most of this overhead, and reduces the training throughput by less than 1%. Note that these overheads are not dependent on the number of nodes, since nodes typically accommodate roughly the same amount of data, bounded by the GPU's HBM storage capacity (i.e., the number of nodes scales with model size). Hence, larger models do not imply higher snapshot creation or tracking latencies.

Checkpoint quantization latency: Quantization is another source of delay. Since checkpoint quantization is done in

dedicated CPU processes (while training continues in GPUs), it does not affect training throughput. However, it introduces a new latency before the checkpoint can be written to storage. For adaptive asymmetric quantization (used by default for 4 bit and lower quantizations), the overhead is determined by the greedy search parameters. Figure 12 depicts the checkpoint quantization latency of adaptive asymmetric quantization as a function of the number of bins used by the greedy algorithm. The latency to quantize is at most 600 seconds even with 50 bins (the bins are described in section 5.2).

Figure 13 shows the checkpoint quantization latency as a function of the ratio used by the greedy algorithm, using 45 and 25 bins. Increasing the ratio requires searching a wider range of the embedding vector values. As such, the latency grows with ratio.

As a comparison, if we only use asymmetric quantization without the adaptation based on bins and ratio, the latency to quantize is at most 126 seconds. Hence, the "adaptive" approach at least doubles the quantization latency.

Note that the above latency values represent the most pessimistic data. But as explained earlier, quantization in Check-N-Run is performed chunk by chunk (as part of the data serialization, where each chunk contains a small subset of the model state). It is pipelined such that each quantized chunk is written independently to the remote storage, while a new chunk is being quantized. Hence, write bandwidth to remote storage is our main bottleneck, and the observed storage write latency is typically higher than the checkpoint quantization latency. Therefore, the latency of our pipelined quantization approach is virtually zero.

6.2 Accuracy

In this section, we evaluate the training accuracy implications of resuming from a quantized checkpoint using the asymmetric and adaptive asymmetric quantizations described earlier. Since differential checkpointing does not alter training accuracy (all data is preserved on every recovery), we focus this section on quantization approaches only. We use a baseline that does not use quantization to determine accuracy loss of quantization.

Note that the number of stored checkpoints and their frequency do not affect the training accuracy, since training is always done in single-precision floating-point. Quantization is only applied to checkpoints, and would only impact the training job if it resumes from a checkpoint. In that case, Check-N-Run would load a checkpoint and de-quantize it before resuming model training in single precision.

When training jobs have to resume from multiple quantized checkpoints during their lifetime, the quantization error may accumulate. Therefore, the number of times a training job resumes from checkpoints determines the suitable quantization bit-width. Figure 14(a) shows the training lifetime accuracy degradation when loading from a 2-bit quantized

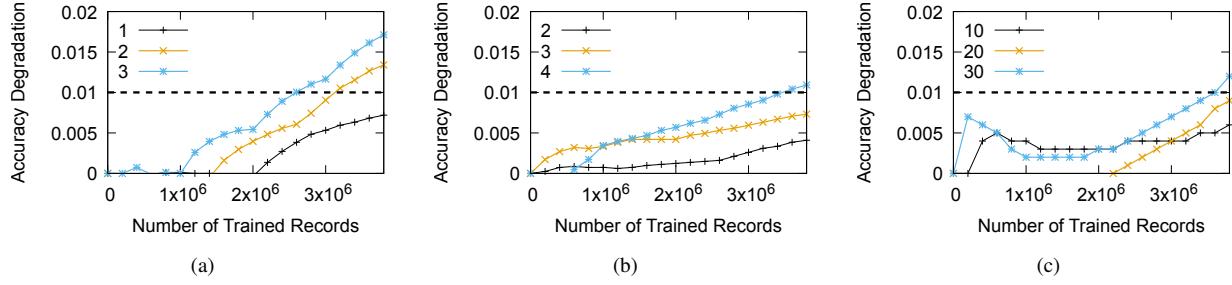


Figure 14: Lifetime accuracy degradation in a training job of 4 billion training samples, when using: (a) 2-bit, (b) 3-bit, and (c) 4bit quantized checkpoints. The lines represent the number of times the job had to resume from a quantized checkpoint

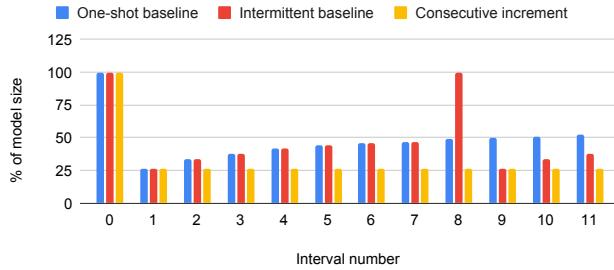


Figure 15: Bandwidth measure: checkpoint size per interval of 30 minutes, for different checkpoint policies

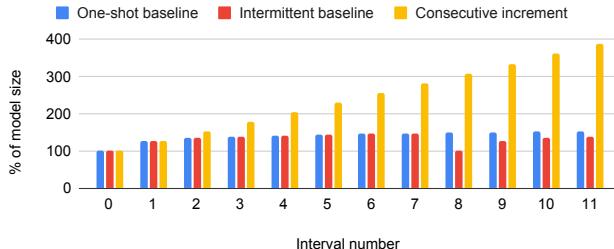


Figure 16: Storage measure: the required storage capacity at each interval of 30 minutes, for different checkpoint policies

checkpoint. We start with 2-bit quantization since it is the most aggressive storage and bandwidth reduction technique of all the approaches. The three lines represent the number of training job failures (failures are uniformly distributed during training), in which the model needs to be reconstructed from a quantized checkpoint. With a single failure, the training accuracy impact is well below the 0.01% threshold even after training with 3 Billion records. However, when two or more failures are encountered during a training run then the 2-bit quantization exceeds the loss threshold of 0.01%.

6.2.1 Dynamic Bit-width Selection:

Figures 14(b) and 14(c) show the accuracy degradation when resuming from 3-bit and 4-bit quantized checkpoints, respectively. As expected, higher bit-widths allow resuming from a checkpoint more times. For 3-bit quantization, a training job may resume from a checkpoint up to 3 times, while for 4-bit quantization one may load the checkpoint up to 20 times. While not shown in the figure, we also measured that with an

8-bit asymmetric quantization, a training job can resume from a checkpoint over 100 times without exceeding the accuracy loss threshold.

Based on the above set of results, Check-N-Run uses a dynamically configurable bit-width selection. Check-N-Run estimates the expected time of training based on the model and the number of nodes. The probability of a node failure in our training cluster (p) is provided as input to Check-N-Run. This probability is computed from failure logs. Check-N-Run then estimates the expected number of failures. Based on this estimate, it picks the bit-width that will not exceed the accuracy threshold. If the number of failures exceeds the estimates during training, Check-N-Run automatically falls back to 8-bit quantization.

6.3 Write Bandwidth and Storage Capacity

In this section, we evaluate the write bandwidth and storage capacity reduction achieved by Check-N-Run, compared with a baseline checkpointing system that uses neither quantization nor differential views.

6.3.1 Differential Checkpointing Policy Comparison

Figure 15 shows the fraction of the model size that is stored in each differential checkpoint, over checkpoint intervals of 30 minutes. This data is a proxy for the bandwidth needed to store the checkpoint. It shows the checkpoint sizes at each interval for different checkpoint policies. In the *One-shot differential* method, the differential checkpoint includes all the embedding vectors that were modified since the first checkpoint, which is created at the first checkpoint interval. As can be seen in the figure, the initial differential checkpoint is only 25% of the total model size, but as the checkpoint size keeps increasing, it exceeds 50% of the model size after 10 intervals. For *Intermittent differential* method, the figure shows how the checkpoint size increases until Check-N-Run dynamically switches to taking a full baseline checkpoint at interval 8, just before the checkpoint size reaches 50% of the model size. The new baseline checkpoint includes the entire model, but the next checkpoint size is only about 25% of the full model size.

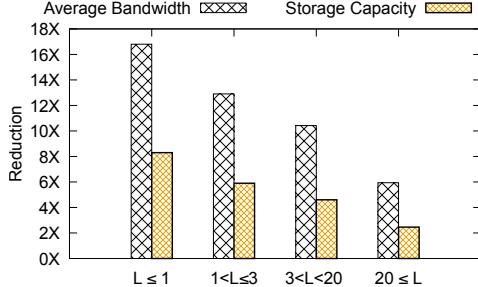


Figure 17: Overall reduction of the checkpoint average write bandwidth and storage capacity. L represents the number of times the training job had to resume from a checkpoint.

Figure 16 shows the total required storage capacity (relative to the model size), over several checkpoint intervals of 30 minutes. The *One-shot differential* approach includes the first checkpoint taken and the latest differential checkpoint at each interval. As expected, the consumed capacity increases over time. The reason is that every differential checkpoint stores *all* the modified entries since the first checkpoint, along side the first checkpoint itself. In the case of *Intermittent differential*, the required capacity increases until the full checkpoint is triggered at interval 8. At that point, the consumed storage capacity resets and includes only the newly taken full checkpoint.

Figures 15 and 16 also show the impact of the *Consecutive incremental* policy, which only stores the vectors that were modified in the current checkpoint interval. The recovery process is more complex, since all previous checkpoints must be read for recovery. As can be seen, this approach reduces the size of checkpoints over time and the corresponding write bandwidth (e.g., the average write bandwidth in a duration of 12 intervals is 33% less than the other policies). Moreover, the checkpoint size is stable, since the number of vectors that are updated during an interval stays roughly the same. However, since all the checkpoints have to be kept, the required storage capacity increases rapidly, reaching almost $\times 4$ the model size after only 11 intervals. As such, Check-N-Run uses the intermittent differential policy by default.

6.3.2 Overall Reduction

Figure 17 presents the overall reduction in write bandwidth and storage capacity, when combining both quantization and differential checkpointing (intermittent baseline policy), and using the thresholds from section 6.2.1 for selecting the quantization bit-width. When a training job is expected to resume from checkpoint no more than one time, Check-N-Run reduces the average consumed write bandwidth and maximum storage capacity by 17 \times and 8 \times , respectively. Even in the not so common case of more than 20 failures, Check-N-Run reduces the average bandwidth by 6 \times and the maximum storage capacity by 2.5 \times . Note that these savings are not linearly proportional to the chosen quantization bit-width due to

the metadata structure. That structure includes the differential checkpoint index and quantization parameters. Metadata structure can be further optimized in future work.

7 Related Work

Checkpointing has been explored in many distributed systems [2, 17, 27, 28, 35]. Checkpoint optimization schemes include techniques to reduce latency [31], coordinating across multiple snapshots for efficient reconstruction [27, 35], using different checkpoint resolutions for providing varying levels of recovery [8, 20]. The goal of Check-N-Run is to deal with checkpoints that are terabytes in size. As such, reducing storage and network bandwidth is important. Unlike traditional distributed systems, where getting a consistent view across different machines is a challenge [2, 28], Check-N-Run exploits the repetitive nature of synchronous training to initiate checkpoints at the end of a training batch.

In terms of ML-specific checkpointing, Deepfreeze [24] checkpoints DNN models using variable resolution, while handling storage-specific API and sharding needs. Microsoft’s ADAM uses zip compression to reduce checkpoint size of DNN models [5]. CheckFreq uses dynamic rate tuning to automatically decide when to initiate a checkpoint and a decoupled store-train pipeline [19]. Check-N-Run tackles reducing storage and bandwidth needs through quantization combined with incremental view. Similar to CheckFreq, it also decouples checkpoint processing from training.

Quantization has been applied to ML models, particularly in the context of inference. Prior works used floating to fixed point quantization to improve compute efficiency [18], ternary quantization for inference on mobile devices [37, 40], per-layer heterogeneous quantization of DNNs [39], mixed precision quantization that adapts to underlying hardware capabilities [34], quantization of gradient vectors for bandwidth efficient aggregation [1, 9, 36], lossy training using 1-bit quantization [29] and more. To the best of our knowledge, using quantization to reduce checkpoint size of recommendation models has not been made public.

8 Conclusion

This paper presents Check-N-Run, a high-performance checkpointing system for training recommendation systems at scale. The primary goal of Check-N-Run is to reduce the bandwidth and storage costs without compromising accuracy. Hence, Check-N-Run leverages differential checkpointing and dynamically selected quantization techniques to significantly reduce the required write bandwidth and storage capacity for checkpointing real-world models. Our evaluations show that depending on the number of recovery events one may need to adapt quantization of different bit widths. By combining such adaptive quantization with differential checkpointing, Check-N-Run provides 6-17x reduction in required bandwidth, while simultaneously reducing the storage capacity by 2.5-8X.

References

- [1] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.
- [2] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.
- [3] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. Revisiting distributed synchronous SGD. *CoRR*, abs/1604.00981, 2016.
- [4] Suming J. Chen, Zhen Qin, Zac Wilson, Brian Calaci, Michael Rose, Ryan Evans, Sean Abraham, Donald Metzler, Sandeep Tata, and Mike Colagrosso. Improving recommendation quality in google drive. In *KDD ’20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 2900–2908. ACM, 2020.
- [5] Trishul M. Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014*, pages 571–582. USENIX Association, 2014.
- [6] Yann Collet and Chip Turner. Smaller and faster data compression with zstandard. <http://www.rgoarchitects.com/Files/fallacies.pdf>, 2016.
- [7] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*, pages 191–198. ACM, 2016.
- [8] Sheng Di, Mohamed-Slim Bouguerra, Leonardo Arturo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 1181–1190. IEEE Computer Society, 2014.
- [9] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 1–8. IEEE, 2016.
- [10] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019. mlsys.org*, 2019.
- [11] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *2015 USENIX Annual Technical Conference, USENIX ATC ’15, July 8-10, Santa Clara, CA, USA*, pages 459–471. USENIX Association, 2015.
- [12] Carlos Alberto Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manag. Inf. Syst.*, 6(4):13:1–13:19, 2016.
- [13] Hui Guan, Andrey Malevich, Jiyan Yang, Jongsoo Park, and Hector Yuen. Post-training 4-bit quantization on embedding tables. In *MLSys Workshop on Systems for ML @ NeurIPS*, 2019.
- [14] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cotter, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook’s dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501. IEEE, 2020.
- [15] Robert L Henderson. Job scheduling under the portable batch system. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294. Springer, 1995.
- [16] Samuel Hsia, Udit Gupta, Mark Wilkening, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. Cross-stack workload characterization of deep recommendation systems. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 157–168. IEEE, 2020.
- [17] R Koo and S Toueg. Checkpointing and recovery rollback for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.
- [18] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*, pages 2849–2858. PMLR, 2016.
- [19] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. Checkfreq: Frequent, fine-grained DNN checkpointing. In Marcos K. Aguilera and Gala Yadgar, editors, *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 203–216. USENIX Association, 2021.

- [20] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC’10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.
- [21] Laurence Morissette and Sylvain Chartier. The k-means clustering technique: General considerations and implementation in mathematica. *Tutorials in Quantitative Methods for Psychology*, 9(1):15–24, 2013.
- [22] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, et al. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. *arXiv preprint arXiv:2003.09518*, 2020.
- [23] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.
- [24] Bogdan Nicolae, Jiali Li, Justin Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. Deep-freeze: Towards scalable asynchronous checkpointing of deep learning models. In *CCGrid’20: 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2020.
- [25] Nvidia. Nvidia hgx2 datasheet. <https://images.nvidia.com/content/pdf/hgx2-datasheet.pdf>.
- [26] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.
- [27] Fabrizio Petrini, Kei Davis, and José Carlos Sancho. System-level fault-tolerance in large-scale parallel machines with buffered coscheduling. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*. IEEE Computer Society, 2004.
- [28] James S Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical report, UT-CS-97-372, 1997.
- [29] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*, pages 1058–1062. ISCA, 2014.
- [30] Brent Smith and Greg Linden. Two decades of recommender systems at amazon.com. *IEEE Internet Comput.*, 21(3):12–18, 2017.
- [31] Nitin H Vaidya. *On checkpoint latency*. Citeseer, 1995.
- [32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.
- [33] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 839–848. ACM, 2018.
- [34] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8612–8620, 2019.
- [35] Long Wang, Karthik Pattabiraman, Zbigniew Kalbarczyk, Ravishankar K Iyer, Lawrence Votta, Christopher Vick, and Alan Wood. Modeling coordinated checkpointing for large-scale supercomputers. In *2005 International Conference on Dependable Systems and Networks (DSN’05)*, pages 812–821. IEEE, 2005.
- [36] Mingchao Yu, Zhifeng Lin, Krishna Narra, Songze Li, Youjie Li, Nam Sung Kim, Alexander G. Schwing, Murali Annavaram, and Salman Avestimehr. Gradiveq: Vector quantization for bandwidth-efficient gradient aggregation in distributed CNN training. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 5129–5139, 2018.
- [37] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.
- [38] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. Aibox: Ctr prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 319–328, 2019.

- [39] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. Adaptive quantization for deep neural network. *arXiv preprint arXiv:1712.01048*, 2017.
- [40] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters

Qizhen Weng^{†*}, Wencong Xiao^{*}, Yinghao Yu^{*†}, Wei Wang[†], Cheng Wang^{*},
Jian He^{*}, Yong Li^{*}, Liping Zhang^{*}, Wei Lin^{*}, and Yu Ding^{*}

[†] Hong Kong University of Science and Technology

^{*} Alibaba Group

{qwengaa, weiwa}@cse.ust.hk, {wencong.xwc, yinghao.yyh, wc189854, jian.h, jiufeng.ly, liping.z, weilin.lw, shutong.dy}@alibaba-inc.com

Abstract

With the sustained technological advances in machine learning (ML) and the availability of massive datasets recently, tech companies are deploying large ML-as-a-Service (MLaaS) clouds, often with heterogeneous GPUs, to provision a host of ML applications. However, running diverse ML workloads in heterogeneous GPU clusters raises a number of challenges. In this paper, we present a characterization study of a two-month workload trace collected from a production MLaaS cluster with over 6,000 GPUs in Alibaba. We explain the challenges posed to cluster scheduling, including the low GPU utilization, the long queueing delays, the presence of hard-to-schedule tasks demanding high-end GPUs with picky scheduling requirements, the imbalance load across heterogeneous machines, and the potential bottleneck on CPUs. We describe our current solutions and call for further investigations into the challenges that remain open to address. We have released the trace for public access, which is the most comprehensive in terms of the workloads and cluster scale.

1 Introduction

Driven by recent algorithmic innovations and the availability of massive datasets, machine learning (ML) has achieved remarkable performance breakthroughs in a multitude of real applications such as language processing [23], image classification [33, 55], speech recognition [32, 56, 62], and recommendation [30, 60, 74]. Today’s production clusters funnel large volumes of data through ML pipelines. To accelerate ML workloads at scale, tech companies are building fast parallel computing infrastructures with a large fleet of GPU devices, often shared by multiple users for improved utilization and reduced costs. These large GPU clusters run all kinds of ML workloads (e.g., training and inference), providing infrastructure support for ML-as-a-Service (MLaaS) cloud [2–4, 7, 8].

In this paper, we share our experiences in running ML workloads in large GPU clusters. We present an extensive

characterization of a two-month workload trace¹ collected from a production cluster with 6,742 GPUs in Alibaba PAI (Platform for Artificial Intelligence) [2]. The workloads are a mix of training and inference jobs submitted by over 1,300 users, covering a wide variety of ML algorithms including convolutional and recurrent neural networks (RNNs and CNNs), transformer-based language models [23, 37, 56], GNNs-based (graph neural network) recommendation models [31, 57, 75], and reinforcement learning [39, 43, 44]. These jobs run in multiple ML frameworks, have different scheduling requirements like GPU locality and gang scheduling, and demand variable resources in a large range spanning orders of magnitude. GPU machines are also heterogeneous (see Table 1) in terms of hardware (e.g., V100, P100, T4) and resource configurations (e.g., GPUs, CPUs, and memory size). In comparison, prior workload analyses focus mainly on training CNN and RNN models in homogeneous environments [18, 29, 36, 41, 65, 66, 72].

The large heterogeneity of ML workloads and GPU machines raises a number of challenges in resource management and scheduling, making it difficult to achieve high utilization and fast job completion. We present those challenges, describe our solutions to some of them, and invite further research on the open problems.

Low utilization caused by fractional GPU uses. In our cluster, a task instance usually can only use parts of a GPU. In fact, the median usage of streaming multiprocessors (SMs) of an instance is 0.042 GPUs. Existing coarse-grained GPU allocation schemes dedicate an entire GPU to one task instance [36, 41, 72], and would result in extremely low utilization in our cluster.

We address this problem with *GPU sharing*, a technique that allows multiple ML tasks to time-multiplex a GPU in a controlled manner [66]. Utilizing this feature, the scheduler consolidates a large volume of low-GPU workloads onto a small number of machines, using only 50% of the requested

¹The trace was collected in July and August 2020, and is now open for public access as part of the Alibaba Cluster Trace Program [1].

GPUs on average. Such consolidation causes no severe interference: among high-utilization GPUs, only 4.5% run ML tasks with potential contention on SMs.

Long queueing delays for short-running task instances. Short-running task instances are prone to long queueing delays caused by head-of-line blocking. In fact, around 9% of short-lived instances spent more than half of their completion time waiting to be scheduled. An effective solution is to predict the task run-time and prioritize short tasks over the long ones. Existing approaches require specialized framework support to track and estimate the training progress [41, 46, 49], which is not always possible in production as users can run standard or customized ML frameworks without such feature.

However, there is a silver lining. In our cluster, the majority of workloads are recurring, with 65% of tasks repeatedly executed at least 5 times in the trace. Through careful feature engineering, we can predict the durations of most recurring tasks within 25% error, sufficient to make quality scheduling decisions as suggested by previous work [16]. Trace-driven simulations shows that using shortest-job-first scheduling with predicted task durations reduces the average completion time by over 63%.

Hard to schedule high-GPU tasks. Our cluster runs a small portion of compute-intensive ML tasks for business-critical, user-facing applications. These tasks request full GPUs (no sharing) and can attain dramatic speedup on high-end devices by exploiting advanced hardware features such as NVLink [12] (see Section 6.1)—these picky requirements make them difficult to schedule.

Our scheduler employs a simple *reserving-and-packing* policy to differentiate those hard-to-schedule high-GPU tasks from other tasks. It reserves high-end GPU machines (e.g., V100 with NVLinks) for a small number of high-GPU tasks with picky scheduling requirements, while packing the other workloads on less advanced machines, using GPU sharing. The reserving-and-packing policy reduces the average queueing delay by 68% for high-GPU tasks and 45% for all.

In our quest for optimized cluster management, a few challenges remain open, which have received less attention in the literature.

Load imbalance. We observe imbalanced load running in heterogeneous machines. In general, machines with low-end GPUs are more crowded than those with high-end GPUs: the former have over 70% CPUs and GPUs of these machines allocated on average, while the latter have only 35% CPUs and 49% GPUs allocated. There is also a *provisioning mismatch* between workloads and machines. On average, workloads running in 8-GPU machines demand $1.9 \times$ more CPUs per GPU than the machines can provide (12 CPUs per GPU), whereas those running in 2-GPU machines request 53% fewer CPUs per GPU than the machine specifications (32 or 48 CPUs per GPU).

Bottleneck on CPUs. While ML workloads perform train-

ing and inference on GPUs, many data processing (e.g., data fetching, feature extraction, sampling) and simulation tasks (e.g., reinforcement learning) involved in the pipeline run on CPUs, which can also become a bottleneck. In fact, we find that workloads running in machines with higher CPU utilization are more likely to get slowdown. For example, in T4 machines, those slowed tasks measure an average of 33.5% P75 CPU utilization, noticeably higher than that measured by the accelerated tasks (21.3%). Similar results are also found in V100 machines reserved for high-GPU workloads (50.6% P75 CPU utilization for slowed tasks and 42.4% for the accelerated), indicating that even GPU-demanding workloads can be harmed by CPU contention.

We believe the observations made in our cluster do not stand in isolation. We share the insights derived from our analysis and discuss potential system optimization opportunities in improving ML framework, adopting resource disaggregation, and decoupling data pre-processing from GPU training (see Section 7). We hope that the observations and experiences shared in our study, as well as the release of the PAI trace, can inspire follow-up research in optimizing ML workload scheduling and GPU cluster management.

2 Background

Fast growing data and GPU demand. The support for scalable machine learning has become increasingly important in production data processing pipelines. In our experience of operating general-purpose ML platforms for production workloads, we have witnessed the fast growing demand of both training data and GPU resources. In just a few years, the sheer volume of training data for an ML job has grown orders of magnitude, from the standard dataset of 100s GB (e.g., ImageNet [22]) to an Internet scale of 10s or even 100s TB. The massive volume of data forces ML jobs to scale out to a large number of GPU machines. In our cluster, the largest single ML job requests to run on over 1,000 GPUs, posing a significant gang-scheduling challenge to the cluster.

Alibaba PAI. To accommodate the fast growing computing demand of ML workloads, Alibaba Cloud offers Machine Learning Platform for AI (PAI), an all-in-one MLaaS platform that enables developers to use ML technologies in an efficient, flexible, and simplified way. PAI provides various services covering the entire ML pipeline, including feature engineering, model training, evaluation, inference, and autoML. Since its introduction in 2018, PAI has gained tens of thousands of enterprises and individual developers, making it one of the largest leading MLaaS platforms in China.

Figure 1 illustrates an architecture overview of PAI, where users submit ML jobs developed in a variety of frameworks, such as TensorFlow [14], PyTorch [48], Graph-Learn [75], RLLib [38]. Upon the job submission, users provide the application code and specify the required compute resources, such

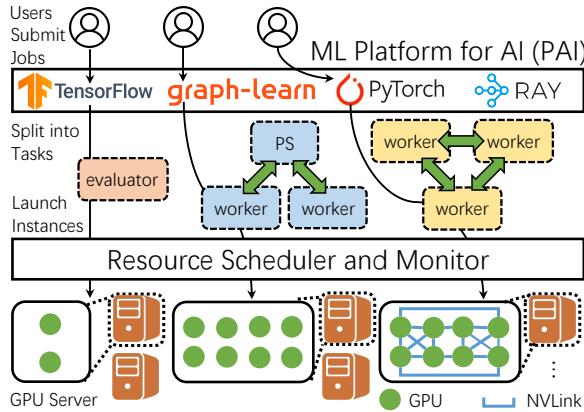


Figure 1: Architecture overview of PAI.

Table 1: Machine specs of GPU clusters in the existing trace analysis works. GPUs with [†] are equipped with NVLink [12]. The Philly trace does not reveal CPU specs and GPU types.

System	#CPUs	Mem (GiB)	#GPUs	GPU type	#Nodes
PAI	64	512	2	P100	798
	96	512	2	T4	497
	96	512	8	Misc.	280
	96	384	8	V100M32 [†]	135
	96	512/384	8	V100 [†]	104
	96	512	0	N/A	83
Philly [36]	Unk.	528/264	2	12GB GPU	321
	Unk.	528/264/132	8	24GB GPU	231
Tiresias [29]	20	256	4	P100 [†]	15
Gandiva _{fair} [18]	12	224	4	K80	32
	12	448	4	P100	12
	12	448	4	V100	6
Themis [41]	24	448	4	K80	12
	12	224	2	K80	8
HiveD [72]	24	224	4	K80	125
	24	224	4	M60	75
Antman [66]	96	736	8	V100M32 [†]	8

as GPUs, CPUs, and memory. Each job is translated into multiple *tasks* of different roles, such as parameter servers (PS) and workers for a training job, and evaluator for an inference job. Each task may consist of one or multiple instances and can run on multiple machines. PAI employs Docker containers to instantiate tasks for simplified scheduling and execution on heterogeneous hardware.

Trace analysis. Running diverse ML workloads in shared GPU clusters at cloud scale raises daunting challenges. Trace analysis is essential to understand those challenges and provide new insights on system optimization. However, existing analyses are performed on GPU clusters with limited size, workload diversity, and machine heterogeneity, and hence cannot fully represent the state of the art (see Table 1). Take Microsoft’s Philly trace [36] as an example. Whereas distributed training is now commonplace, the majority of Philly

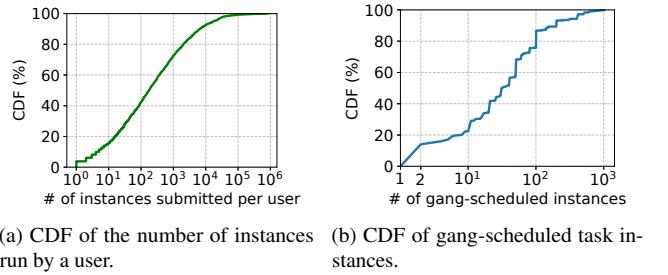


Figure 2: Heavily skewed distribution of task instances run by users and the prevalence of gang-scheduling requirements.

workloads (> 82%) ran on a single GPU instance when the trace was collected in 2017. It is also unclear what types of GPUs were used to run those workloads, which may have significant impact to scheduling [41, 46]: the performance of new-generation GPUs can be 1.1–8× higher than the older generations [18]. Moreover, the Philly trace only includes the training workloads, whereas it is common to run both training and inference jobs in a shared MLaaS platform [47, 51, 69].

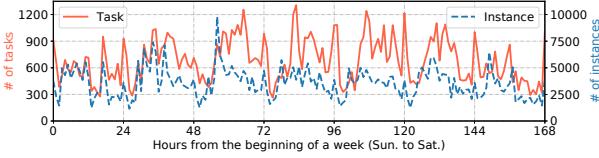
The insufficiency of existing works motivates the release of the PAI trace, which we examine next.

3 Workload Characterization

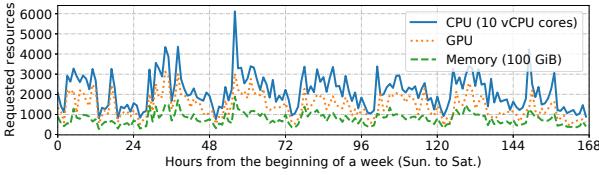
In this section, we analyze the ML workloads in the released PAI trace. We start with an overview of the trace, followed by a characterization of its temporal and spatial patterns.

3.1 Trace Overview

Trace information. The released PAI trace contains a hybrid of training and inference jobs running state-of-the-art ML algorithms in mainstream frameworks [14, 48, 75]. Most jobs request multiple GPUs. The trace records the arrival time, completion time, resource requests and usages in GPUs, CPUs, GPU memory and main memory of the workloads at various levels (e.g., job, task, and instance) (Sections 3.2 and 3.3). The application semantics, such as whether the code is performing training or inference, and in what ML framework, are not available as our cluster scheduling system Fuxi [26, 71] only sees the execution containers and is agnostic to the running applications. Nevertheless, we have manually examined some workloads and included their application names (e.g., click-through rate prediction and reinforcement learning) in the trace to provide some clues whenever possible (Sections 6.1 and 6.2). Machine-level information is also provided in the trace, including the hardware specs (Table 1) and time-varying resource utilizations (Section 4) collected by the daemon agents that periodically query the Linux kernel and GPU driver (e.g., NVIDIA Management Library [9]) in the host machines. The detailed schema and trace data are given in the trace repository [1].



(a) Number of tasks submitted and their instances in one week.



(b) Total resource requests of running tasks in one week.

Figure 3: Task submissions and resource requests roughly follow diurnal patterns.

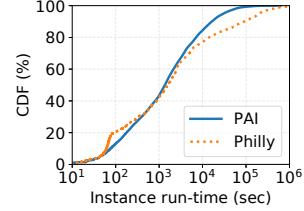
Jobs, tasks, and instances. In PAI, users submit *jobs*. Each job has one or multiple *tasks* taking different computation roles. Each task runs one or multiple *instances* in Docker containers. For example, a distributed training job may have a parameter-server (PS) task of 2 instances and a worker task of 10 instances. All instances of a task have the same resource demands and might be *gang-scheduled* (e.g., running simultaneously for all PyTorch workers). Our characterization in this subsection mainly focuses on task instances.

Heavy-skewed instance distribution. The PAI trace contains more than 7.5 million instances of 1.2 million tasks submitted by over 1,300 users. Figure 2a depicts the distribution of task instances run by users, which is *heavily skewed*. More specifically, around 77% of task instances are submitted by the top 5% users, each running over 17.5k instances, while the bottom 50% users run less than 180 instances each.

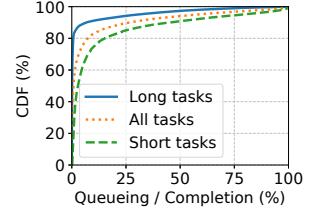
The prevalence of gang-scheduling. Our distributed ML jobs require gang-scheduling. As shown in Figure 2b, among all task instances, around 85% have such requirements, in which 20% must be gang-scheduled on more than 100 GPUs, some even requesting over 1,000. Together, tasks with gang-scheduled instances account for 79% of the total GPU demands. The prevalence of these tasks makes it difficult to achieve high utilization.

GPU locality. In addition to gang-scheduling, a task may request to run all its instances on multiple GPUs co-located in one machine, a requirement known as *GPU locality*. Although such requirement often leads to prolonged scheduling delays [29, 36, 72], it enables the use of high-speed GPU-to-GPU interconnect within a single node (e.g., NVLink and NVSwitch), which can dramatically accelerate distributed training [12, 15, 36]. In our cluster, enforcing GPU locality yields over 10 \times speedup for some training tasks (Section 6.1).

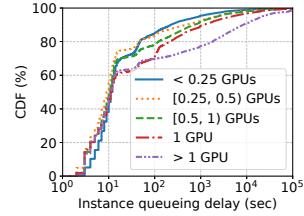
GPU sharing. PAI supports *GPU sharing* that allows multiple task instances to time-share a GPU at a low cost. With this feature, users can specify GPU request in (0, 1) and run



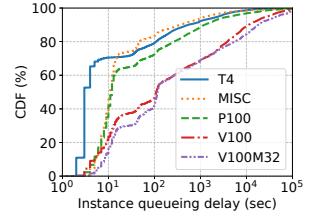
(a) CDF of instance run-time.



(b) CDF of normalized instance queueing delays.



(c) CDF of queueing delays w.r.t. GPU requests per instance.



(d) CDF of queueing delays w.r.t. GPU types.

Figure 4: CDF of instance run-time and queueing delays.

its task instances using parts of GPUs. We will show in Section 5.1 that GPU sharing enables considerable savings on GPU provisioning.

Various GPU types to choose from. PAI provides heterogeneous GPUs and allows users to specify the required GPU types to run their tasks. The available choices include NVIDIA Tesla T4, P100, V100, V100M32 (V100 SXM2 with 32 GB memory), and other GPUs of older generations (Misc in Table 1), e.g., Tesla K40m, K80, and M60. In our cluster, only 6% tasks require to run on specified GPUs, while the others have no such limitation and can run on any GPUs.

3.2 Temporal Pattern

We next examine the temporal patterns of the PAI workloads.

Diurnal task submissions and resource requests. Figure 3 depicts task and instance submissions as well as the overall resource requests in one week during the trace collection period. We observe rough diurnal patterns, where task submissions in weekdays (from the 24th to 144th hours) are slightly higher than in weekends. It is worth mentioning that in addition to the daytime, midnight is also a rush hour for task submissions (Figure 3a). Yet, most tasks submitted at midnight are less compute-intensive, having only a few instances and requesting a small amount of resources (Figure 3b).

Instance run-time in a wide range. Figure 4a shows the distribution of instance run-time (solid line). Similar to the Philly trace [36] (dotted line), instance run-time varies in a wide range spreading four orders of magnitude. The median run-time (23 minutes) is comparable with that of Philly (26 minutes), while their 90th percentile (P90) run-time (4.5 hours) is shorter than that of Philly (25 hours).

Non-uniform queueing delays. The queueing delay (aka wait time or scheduling delay), measured from the moment of task submission to the start of the task instance, varies greatly among instances. Compared to the long-running instances, short-running instances usually spend a larger portion of time in queueing. To see this, we use the median run-time as a threshold and divide instances into long-running and short-running ones, where a long-running (short-running) task instance has a longer (shorter) run-time than the median. In Figure 4b, We compare the queueing delays of these task instances relative to their completion times (queueing delay plus run-time). Around 9% short-running instances spend more than half of the completion time waiting to be scheduled; this number drops to 3% when it comes to long-running instances.

A task instance’s queueing delay also depends on its GPU request. Figure 4c shows that instances willing to share GPUs (i.e., GPU request in $(0, 1)$) can be quickly scheduled, with the 90th percentile (P90) queueing delay being 497 seconds. In comparison, instances that do not accept GPU sharing need to wait for a longer time, with the P90 delay being 1,150 (8,286) seconds for those requesting one GPU (> 1 GPU).

Long queueing delays are also seen in instances requesting high-end GPUs. As shown in Figure 4d, for instances running on advanced V100 GPUs (including V100M32), the median and P90 delays are 113 and 13,709 seconds, respectively. In comparison, for instances running on low-end miscellaneous GPUs, the median and P90 delays are only 11 and 360 seconds, respectively.

3.3 Spatial Pattern

We finally present the spatial patterns of the PAI task instances by analyzing their resource requests and usages. PAI collects the system metrics of running tasks every 15 seconds and provides visualization tools [2, 25] for users to analyze the workload patterns and figure out their resource requests.

Heavy-tailed distribution of resource requests. Figures 5a, 5b, and 5c (blue solid lines) respectively depict the distributions of the total CPUs, GPUs, and memory requested by all instances. All three distributions are heavy-tailed, with around 20% instances requesting large resource amounts and the other 80% requesting small to medium. More specifically, the P95 request demands 12 vCPU cores², 1 GPU, and 59 GiB memory, more than twice the median request (6 vCPU cores, 0.5 GPUs, and 29 GiB memory).

Uneven resource usage: Low on GPU but high on CPU. Most users tend to ask for more resources than they actually need, resulting in a low resource usage (dotted lines in Figures 5a, 5b, and 5c). In our cluster, the median instance resource usages are 1.4 vCPU cores, 0.042 GPUs, and 3.5 GiB memory, much smaller than the median request. We stress

²In our cluster, each physical processor core consists of two vCPU cores, using hyper-threading technology [42].

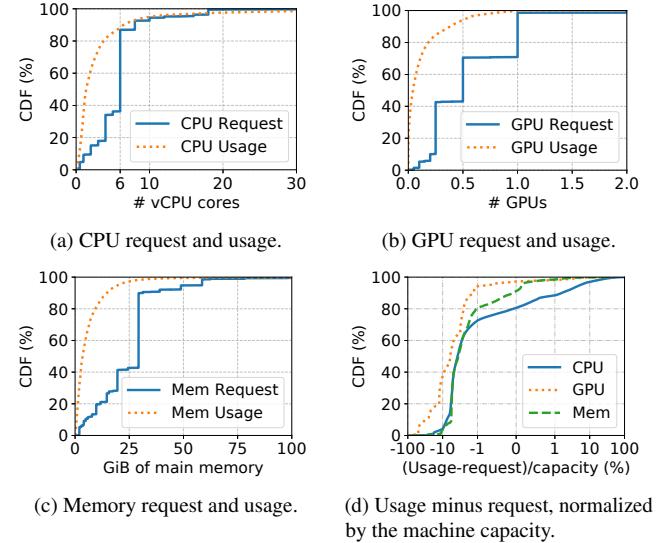


Figure 5: CDF of instance resource requests and actual usages.

that the low GPU usage is not caused by the low computing demand, but by contentions on other resource like CPU, making GPUs idle for most of the time (Section 6.2). Figure 5b also shows that around 18% instances barely use GPUs: they perform computations such as running parameter servers, fetching and pre-processing data, which are mostly on CPUs with small or no GPU involvement.

In PAI, instances of a task can use *spare resources* in the host machines, making it possible to *overuse* more resources than requested. Compared to GPU and memory, overuse of CPUs is more prevalent. To see this, for each instance we measure the difference between its resource usage and request for CPU, GPU, and memory—positive (negative) being overuse (underuse). We normalize the results by the machine’s CPU, GPU, and memory capacity, respectively, and depict the distributions in Figure 5d. There are 19% task instances overusing CPUs (blue solid line with $X > 0$). In comparison, only 3% (9%) instances use more GPUs (memory) than they requested.

4 GPU Machine Utilization

Having studied the workload characterization, we turn to resource utilization in GPU machines.

4.1 Utilization of Compute Resources

We start to analyze the utilization of compute resources, including CPU, GPU, main and GPU memory. Our cluster has 1,295 2-GPU machines and 519 8-GPU machines (Table 1). Machines with 8 GPUs have a lower CPU-to-GPU ratio than those with 2 GPUs. In light of their different configurations, we perform measurement separately for the two types of machines. Each machine has time series data of resource utilization measured every 15 seconds by the monitoring system.

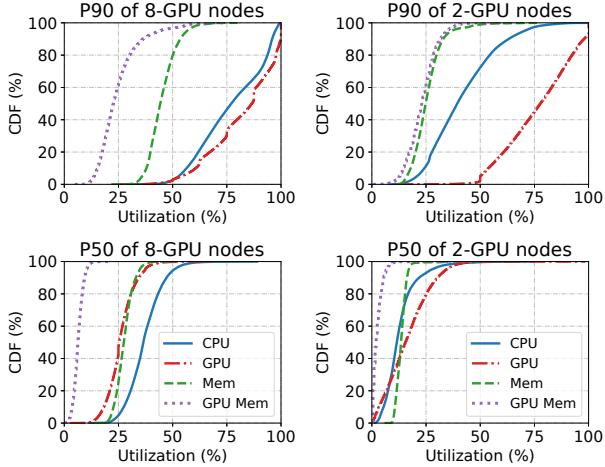


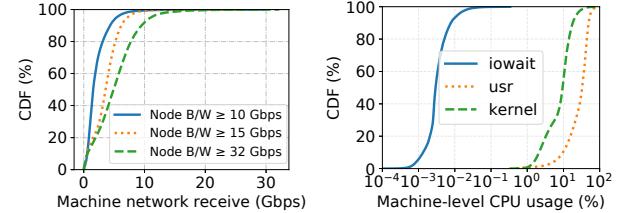
Figure 6: CDF of P90 and P50 (median) utilization of CPU, GPU, main and GPU memory in different machine groups.

At each timestamp, we collect the utilization of all 8-GPU machines and calculate the tail (P90) and the median (P50). Together, we obtain a sequence of P90 and P50 utilizations taken at different timestamps. We depict their distributions in Figure 6 (two subfigures on the left). We perform the same measurements in 2-GPU machines and depict the results in the right two subfigures. Compared to memory (main and the GPU’s), GPU and CPU have higher utilization. In 8-GPU machines (upper-left in Figure 6), the average P90 utilization of GPU (red dash-dotted line) and CPU (blue solid line), i.e., the arithmetic mean of P90 values from all timestamps, reaches 82% and 77%, respectively. In 2-GPU machines (upper-right in Figure 6), the P90 GPU utilization remains high (77% on average), while the P90 CPU utilization drops to 42% on average due to the large CPU-to-GPU ratio (32 or 48 CPUs per GPU). In both types of machines, the P90 utilization of the main and GPU memory stays below 60% at almost all time, indicating that our tasks are less memory-intensive.

Compared to other resources, we measure a larger variation of utilization on GPUs. As shown in Figure 6, the distribution of P90 GPU utilization spans a wide range from less than 40% to 100% of the computing power provided by the streaming multiprocessors of the machine’s GPUs; the difference between the tail and the median utilization is also larger on GPU than on other resources (comparing the top sub-figures with the bottom). The large variation is partly due to the bursty GPU usage patterns found in our ML workloads [65, 66]. It is also due to the design of our scheduler that prioritizes packing over load balancing (Section 6.3).

4.2 Low Usage of Network and I/O

In addition to compute resources, network and I/O are also frequently used in distributed ML. To understand their impact,



(a) CDF of machine network input. (b) CDF of machine CPU time.

Figure 7: Low usage of network and I/O.

we measure the network input rate³ in machines with different bandwidth guarantees (≥ 10 Gbps for P100 and Misc, ≥ 15 Gbps for T4, and ≥ 32 Gbps for V100) and depict their distributions in Figure 7a. The P95 network input rate only reaches 54%, 48%, and 34% of the guaranteed bandwidth provided in P100 (or Misc), T4, and V100 machines, respectively.

In terms of I/O, we collect machine-level CPU usage data, including the I/O waiting time (iowait) and the execution time in usr and kernel modes, respectively. Figure 7b shows their distributions. The CPU time spent on iowait is three orders of magnitude smaller than that in usr and kernel modes, meaning that CPUs are mostly busy processing data rather than waiting for the I/O to complete.

5 Opportunities for Cluster Management

In PAI, our goal of cluster management is two-fold: (1) achieving high utilization in GPU machines, and (2) completing as many tasks as fast as possible. In this section, we describe the opportunities and our efforts in achieving the two goals.

5.1 GPU Sharing

Unlike CPUs, GPUs do not natively support sharing and are allocated as indivisible resources in many production clusters [36, 72], where a single task instance runs exclusively on a GPU. Although such allocation provides strong performance isolation, it results in GPU underutilization, which is particularly salient in our cluster as most instances can only utilize a small portion of the allocated GPUs (Section 3.3).

To avoid this problem, the PAI cluster scheduler supports GPU sharing which allows multiple task instances to run on the same GPU in a space- and time-multiplexed manner. With this feature, a task instance can request a fraction of GPU (< 1 GPU) and is guaranteed to allocate the specified fraction of GPU memory upon scheduling (space-multiplexed). When needed, an instance can also use unallocated GPU memory during execution. An instance, however, has no guaranteed allocation of compute units (i.e., SMs), which are dynamically shared among co-located instances (time-multiplexed).⁴

³Our trace does not log the network output. For most training and inference tasks, the network input is orders of magnitude larger than the output.

⁴Fine-grained sharing of compute units with isolation guarantee requires

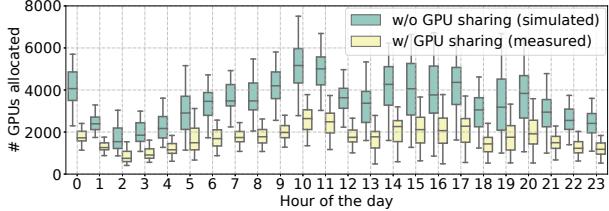


Figure 8: Box plots of the number of allocated GPUs with and without GPU sharing. The boxes depict the 25th, 50th, and 75th percentiles; the two whiskers are one interquartile range (IQR) past the low and high quartiles.

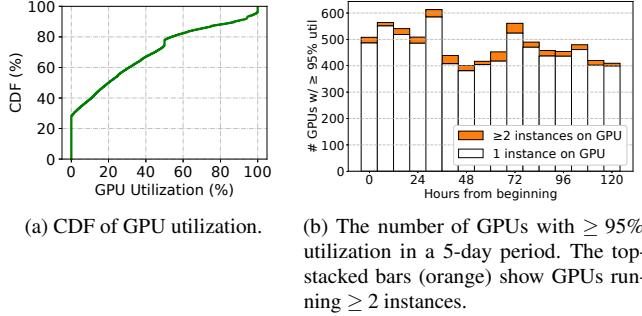


Figure 9: Heavy utilization is rarely measured in GPUs; most heavy-utilized GPUs run a single instance.

Benefits of GPU sharing. GPU sharing enables considerable savings on resource provisioning. To see this, we simulate the scenario of no GPU sharing, in which we replay the trace and count the number of allocated GPUs in each hour. Figure 8 compares the simulated results with the numbers measured in the real system, binned in hour of the day. On average, only 50% of GPUs are needed with sharing. In the peak hour at around 10 am, the savings can be up to 73%.

Does GPU sharing cause contention? As the utilization increases, instances running on a shared GPU start to contend for streaming processors (SMs), causing interference. To quantify how frequently the contention may occur, we collect the utilization data of all GPUs in two months and depict their distribution in Figure 9a. Heavy utilization ($\geq 95\%$) is rarely measured, which accounts for only 7% cases in the trace. We further examine those heavy-utilized GPUs in which running instances have a high chance to contend with each other. Figure 9b shows the number of heavy-utilized GPUs in a 5-day period, among which only a few (4.5% on average) run multiple instances (the top-stacked bars). As the majority of heavy-utilized GPUs run a single instance, no contention occurs. We therefore believe GPU sharing does not cause severe contention in our cluster.

high-level support of ML framework. In PAI, such support is provided by AntMan [66]. Yet, it only applies to tasks running in the frameworks where AntMan is implemented (currently supporting TensorFlow and PyTorch).

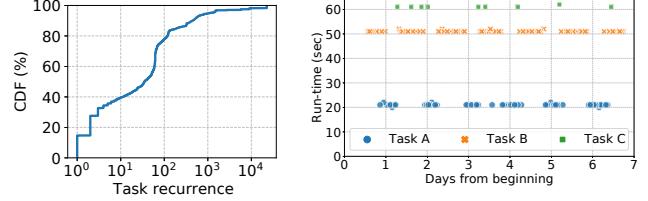


Figure 10: CDF of task recurrence.

Figure 11: Submissions and instance run-times of three batch inference tasks using BERT.

5.2 Predictable Duration for Recurring Tasks

Knowing the duration (aka run-time) of ML task instances is the key to making better scheduling decisions. Existing schedulers for ML workloads predict the task instances duration based on the training progress (e.g., number of iterations, loss curve, and target accuracy) and speed of the task [29, 41, 46, 49]. Obtaining such information requires specific framework support (e.g., TensorFlow and PyTorch), which is not always possible in our cluster as users run a variety of frameworks of standard or customized version, and their submitted tasks may not perform iterative training (e.g., inference). In fact, our cluster scheduler [26, 71] is designed for container workloads and is agnostic to the task semantics.

The prevalence of recurring tasks. Despite the scheduler being agnostic to task progress, we find that most tasks are *recurring*, and their instance run-times can be well predicted from past executions. Yet, in our system, task recurrence cannot be simply identified from the task ID or name, which is uniquely generated for each submission. Instead, we turn to the meta-information consistently specified by a task across multiple submissions, such as the entry scripts, command-line parameters, data sources and sinks. Hashing the meta-information generates a unique Group tag, which we use to identify the recurrence of a task. Following this approach, we depict the distribution of task recurrences in Figure 10: around 65% tasks repeatedly run at least 5 times in the trace.

In addition to periodic training, many recurring tasks perform *batch inference*. These tasks aggregate data from incoming requests and then perform batch inference on a collective of data in one go. Users can configure the task launching interval, ranging from minutes to days. As an illustrative example, Figure 11 shows three recurring tasks identified in the trace that perform batch inference with pre-trained BERT [23] models. All three tasks run on a regular basis, with stable average instance run-times that can be accurately predicted.

Instance duration prediction for recurring tasks. A recurring task can be submitted by different users with different resource requests, and its instances may have different run-times. We therefore predict the duration from past runs based on three features, the task’s username (User), resource requests (Resource, including GPU and other resources), and group tag (Group). Taking these features as input, we predict

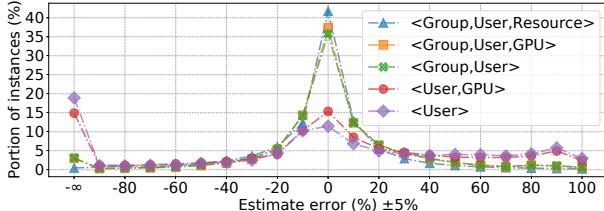


Figure 12: Percentage prediction error, i.e., $(\text{true} - \text{pred})/\text{true}$ in percentage, of duration estimates with different features.

the task’s average instances duration using the CART (Classification And Regression Trees [17]) algorithm with a tree regressor. The regressor makes at most 10 splits for each tree and uses the mean absolute error (MAE) as the splitting criterion. We choose MAE instead of the standard mean squared error (MSE) because the former is more robust to extreme outliers in heavy-tailed distribution than the latter.

To evaluate the accuracy of our prediction, we consider tasks that recur at least 5 times in the trace. We use 80% of those tasks to train the predictor and the remaining 20% for testing. Figure 12 compares the accuracy of the predictor trained with different feature inputs, including Group, User, Resource, and GPU (requested GPU types and numbers). We use percentage prediction error [35] as the accuracy metric, defined as $(\text{true} - \text{pred})/\text{true} \times 100\%$. Our evaluation shows that Group is the most important feature that greatly improves the prediction accuracy. Further complementing it with User and Resource (or GPU) results in less than 25% prediction error for 78% instances. According to prior studies [16], duration predictions with such accuracy is sufficient to make high-quality scheduling decisions.

Benefits for scheduling. We present a simple simulation study to evaluate how the prediction of task instance duration can help improve scheduling. We developed a discrete-time simulator and use it to replay the trace. We sample tasks from the trace and feed their resource requests, arrival times, real and predicted run-times into the simulator. We assume homogeneous GPUs in simulation and respect the real duration when scheduling a task instance to a GPU. Both the simulator and experiment scripts are released along with the trace [1].

We configure two scheduling policies, first-in-first-out (FIFO) and shortest-job-first (SJF), in simulation. Figure 13 shows the average task completion time in GPU clusters of different sizes using FIFO and four SJF schedulers, where SJF-Oracle makes scheduling decisions based on the real-measured task instance duration (ground truth) and the others use predictors trained with different input features. Compared to FIFO, the four SJF schedulers reduce the average task completion time by 63–77%, depending on the predictors they use. In particular, the predictors trained with the Group feature yield better performance; the more features are included, the more accurate the predictions are, and the closer the scheduling performance is to the optimum (SJF-Oracle).

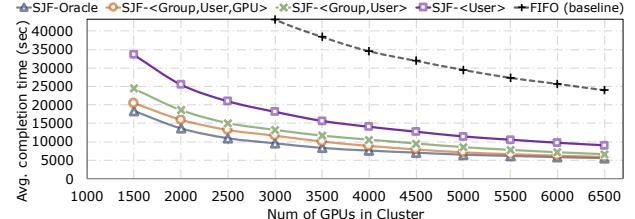


Figure 13: Average task completion time given different GPU cluster sizes and various scheduling policies in simulation.

These results are in line with Figure 12.

6 Challenges of Scheduling

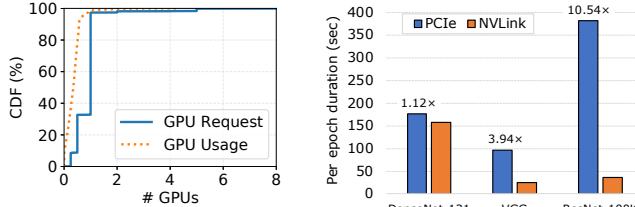
Compared to previous simulations, scheduling ML tasks of large heterogeneity in production clusters is far more complex. To understand the challenges posed by such heterogeneity, in this section we present case studies for two representative types of ML tasks with high and low GPU requests. We describe our scheduling policies deployed in production that differentiate between the two types of tasks in light of their different request and usage patterns. Yet, many challenges remain open, which we discuss in detail.

6.1 Case Study of High-GPU Tasks

In our cluster, a small portion of tasks run compute-intensive instances with high GPU requests (Section 3.3). These tasks train state-of-the-art models or perform inference with trained models for business-critical, user-facing applications. They request powerful GPU devices with high memory or advanced hardware features (e.g., NVLink).

NLP with advanced language models. Around 6.4% tasks running in our cluster perform natural language processing (NLP) using advanced models, such as BERT [23], ALBERT [37], and XLNet [67]. Among them, 73% have large input and must run on GPUs with 16 GiB or higher memory (i.e., T4, P100, V100/V100M32). Figure 14a shows the distribution of GPU requests and usages of NLP instances, where 40% request more than 1 GPU and use over 0.4 GPUs in computing power. Comparing Figure 5b and Figure 14a, we observe much higher GPU requests and usages of NLP tasks than that of general workloads.

Image classification with massive output. In our cluster, some distributed training tasks request to run their worker instances in one machine with high-speed GPU-to-GPU interconnects (e.g., NVLink) for much improved performance, a requirement known as *GPU locality*. A typical example is to train a classification model that classifies images of goods into a large number of standard product units (SPUs). The model can be a modified ResNet [33] with the last output layer replaced by a softmax layer with 100,000 output of SPUs



(a) CDF of GPU requests and usages of NLP task instances.

(b) Per-epoch duration of 3 classification models trained in 8-GPU machines with and without NVLink.

Figure 14: High-GPU tasks (NLP and image classification).

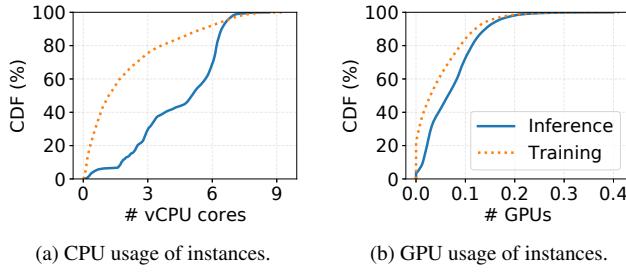


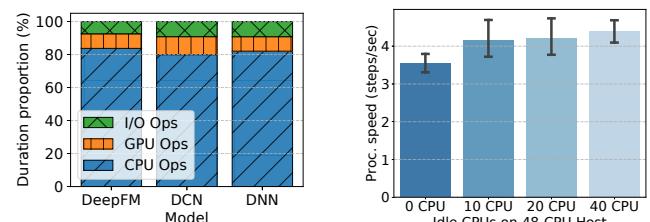
Figure 15: CDF of the CPU and GPU usage of click-through-rate (CTR) instances.

(ResNet-100k). The presence of such a large fully-connected layer mandates the exchange of massive gradient updates between worker instances, making communication a bottleneck. For these tasks, meeting GPU locality is critically important. Figure 14b compares the duration of a training epoch of three classification models with a large number of output in 8-GPU machines with and without NVLink (i.e., via PCIe). All three models achieve salient speedup with NVLink: ResNet-100k, the largest model, is accelerated by 10.5 \times .

6.2 Case Study of Low-GPU Tasks

The majority of tasks running in our cluster have low GPU requests and usages (Section 3.3). To understand this somewhat unexpected result, we study three popular tasks. By profiling their executions, we find that they spend a considerable amount of time on CPUs for data processing (e.g., data fetching, feature extraction, sampling) and simulation (e.g., reinforcement learning), leaving GPUs under-utilized.

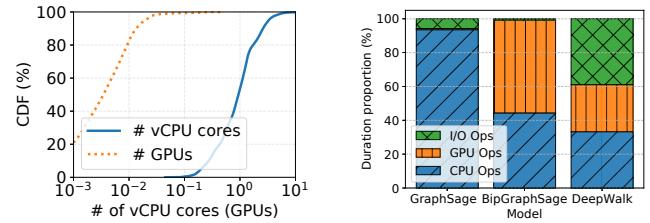
CTR prediction model training and inference. Among all tasks in the trace, over 6.7% are for advertisement click-through rate (CTR) prediction. These tasks use a variety of CTR models [30, 60, 73, 74], with around 25% instances performing training and the other 75% performing inference. Figure 15 shows the distributions of the CPU and GPU usages of these instances. Compared to training, inference instances have higher CPU utilization as they process a large volume of data continuously arriving. Both instances have low GPU utilization: over 75% instances use less than 0.1 GPUs.



(a) Duration breakdown of CTR prediction instances.

(b) DeepFM training instances interfered by the co-located load.

Figure 16: Microbenchmark of inference and training instances of click-through-rate prediction models.



(a) CDF of CPU and GPU usage.

(b) Instance duration breakdown.

Figure 17: Resource usage and duration breakdown of GNN training instances.

We next profile the executions of three inference instances with DeepFM, DCN, and DNN models, respectively. Figure 16a shows the run-time breakdown of I/O, GPU, and CPU operations. The three instances spend around 80% runtime on CPUs to fetch and process the next input batch (IteratorGetNext in TensorFlow [20, 40]); GPU and I/O operations (e.g., MatMul, Sum, Cast, MEMCPYHtoD) only account for 10% of the execution time, respectively.

The high CPU usage of these instances makes them prone to interference from the co-located workload, especially in machines with high CPU utilization. To see this, we run training instances of a DeepFM model in containers with 8 vCPU cores. Together with an instance, we run some artificial load using spare cores of the host machine to create CPU stress. We configure varying load to control the level of stress. Figure 16b shows the instance training speed in a 48-core machine under varying stresses with 0 to 40 cores left idle (highest to no stress). Though the co-located load run on different vCPU cores not occupied by the instance, it still results in up to 28% slowdown of the training speed due to the contention of other shared resources, such as cache, power, and memory bandwidth [19, 21, 58].

GNN training. Graph Neural Network (GNN) training comes as another popular computation, which accounts for 2% instances in our cluster, including GraphSage [31], Bipartite GraphSage [75], GAT [57], etc. Figure 17a shows the distribution of CPU and GPU usage of GNN training instances, where CPU is more heavily utilized than GPU. In production GNN models, a graph must undergo a sequence

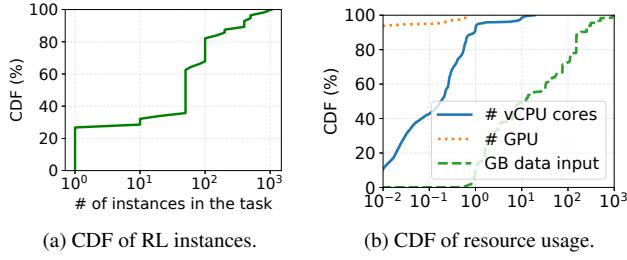


Figure 18: Characterization of reinforcement learning instances.

of pre-processing, such as EdgeIteration, NeighborSampling, and NegativeSampling [75], before turning into an embedding (a computationally digestible format, usually vectors) of a deep neural network. Such graph pre-processing is currently cost-effective when performing on CPUs. As shown in Figure 17b, it accounts for 30–90% duration of each training iteration in different models.

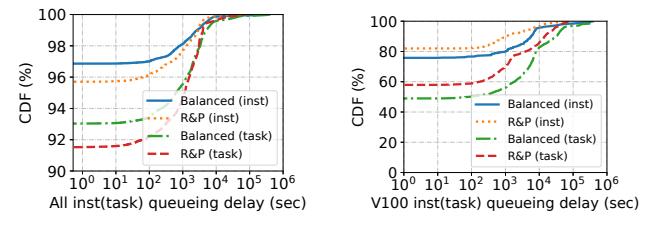
Reinforcement learning. Our cluster also runs many reinforcement learning (RL) tasks. An RL algorithm iteratively generates a batch of data through parallel simulations on CPUs and performs training with the generated data on GPUs to improve the learning policy. Figure 18a shows that 72% RL tasks have at least 10 gang-scheduled instances, with the largest one running over 1,000 instances. Most RL instances are used to run simulations, eating up lots of CPUs and network bandwidth but only a small fraction of GPUs, as shown in Figure 18b. In fact, in the largest RL task, each instance requests only 0.05 GPUs.

6.3 Deployed Scheduling Policies

Compared to low-GPU tasks, high-GPU tasks have picky scheduling requirements and are usually run by business-critical applications. They are hence differentiated from other tasks and scheduled as first-class citizens.

Reserving-and-packing. In our cluster, the scheduler employs a *reserving-and-packing* policy. That is, it intentionally reserves high-end GPUs (e.g., V100/V100M32 with NVLinks) for high-GPU tasks, while packing the other workloads to machines with less advanced GPUs (e.g., T4 and Misc). Specifically, for each task, the scheduler characterizes its *computation efficiency* using a performance model that accounts for many task features, such as the degree of parallelism, the used ML model, the size of embedding [59, 64, 70], and the historical profiles of other similar tasks. Tasks with high computation efficiency larger than a certain threshold are identified as high-GPU.

For each task, the scheduler generates an ordered sequence of *allocation plans*; each plan specifies the intended GPU device and is associated with an attempt timeout value. The scheduler attempts allocation following the ordered plans: it waits for the availability of the intended GPU specified in the



(a) Queueing delays of all instances and tasks.
(b) Queueing delays of V100 instances and tasks.

Figure 19: Task queueing delays in simulation with load-balancing (Balanced) and reserving-and-packing (R&P).

current plan until timeout, and then moves on to the next plan for another attempt. For high-GPU tasks, the allocations of high-end GPUs are attempted before the less advanced ones in the ordered plans; for other tasks, the order is reversed. Our GPU scheduler is implemented atop Fuxi [26, 71], a locality-tree based scheduling system.

Load-balancing. Given the potential resource contention and interference between co-located task instances (Section 6.2), maintaining load balancing across machines with similar specs is also important. Therefore, under reserving-and-packing, the scheduler also prioritizes instance scheduling to machines with low *allocation rate*, measured as a weighted sum of the allocated CPUs, memory, and GPUs normalized by the machine’s capacity.

Benefits. Our scheduler prioritizes reserving-and-packing over load-balancing. To justify this design, we evaluate two scheduling policies using the simulator described in Section 5.2: ① simply load-balancing machines using progressive filling (always scheduling a task’s instances to the least utilized node), and ② only performing reserving-and-packing without considering load balancing (R&P). We sample 100,000 tasks with over 500,000 gang-scheduled instances from the trace and feed them into the simulator. Figure 19a shows the CDF of the queueing delays of all instances and tasks under the two policies. Note that the queueing delay of a task is also the queueing delay of its gang-scheduled instances. Over 90% instances and tasks are launched immediately under the two policies. Compared to load-balancing, reserving-and-packing reduces the average task queueing by 45%, mostly attributed to the significant cutoff of the tail latency by over 10,000 seconds. Figure 19b further compares the queueing delays of business-critical tasks and instances requesting V100 GPUs under the two policies: reserving-and-packing reduces the average task queueing delay by 68%. The simulation results justify our design of prioritizing reserving-and-packing over load-balancing.

6.4 Open Challenges

However, our scheduler policy design is not without its problems, many of which remain open to address. We next discuss

Table 2: Mismatch between machine specs and instance requests, in terms of the provisioned/requested CPUs per GPU.

vCPU cores per GPU	All nodes	8-GPU nodes	2-GPU nodes
Machine specs	23.2	12.0	38.1
Instance requests	21.4	22.8	18.1

those open challenges, which we believe also stand in other GPU clusters with heterogeneous machines.

Mismatch between machine specs and instance requests. We observe a mismatch between machine specs and instance requests. Table 2 compares the average number of provisioned and requested vCPU cores per GPU in machines with 8 and 2 GPUs and their running instances. In 8-GPU machines, 12 vCPU cores are provisioned for each GPU. Yet, the instances running in those machines request 22.8 vCPU cores per GPU on average. On the other hand, CPUs in 2-GPU machines are over-provisioned, where the CPU-to-GPU ratio is more than twice of the instance requests.

To understand how the mismatch may affect the machine utilization, we randomly sample a number of nodes with different specs and depict the requests and usages of CPUs and GPUs in heatmaps shown in Figure 20, where each row corresponds to one machine, and all values are normalized to the machine’s capacity. Compared to 8-GPU nodes, 2-GPU machines have substantially underutilized CPUs despite GPUs being heavily occupied. On average, P100 (T4) machines have 31% (20%) CPUs allocated with only 19% (10%) CPU utilization (Figures 20c and 20d).

We stress that the mismatch between machine specs and instance requests is not fundamental, as the cluster-wide CPU-to-GPU specs remains close to the overall instance requests (23.2 vs. 21.4 as shown in Table 2). We therefore believe that the mismatch can be avoided or at least mitigated by improved scheduling (e.g., rescheduling some high-CPU instances in 8-GPU machines to 2-GPU nodes).

Overcrowded weak-GPU machines. Compared to other machines, those with less advanced GPUs are overcrowded. The problem becomes even more salient in 8-GPU nodes (Misc GPUs) as shown in Figure 20a. On average, 77% CPUs and 74% GPUs are allocated in these machines. CPUs are better utilized than GPUs: the utilization of CPU is 43% on average, while the average utilization of GPU is 18%. This result is partly caused by our scheduling algorithm prioritizing weak-GPU machines for low-GPU tasks (Section 6.3), which account for a large instance population in our cluster.

Imbalanced load in high-end machines. Compared to other nodes, high-end machines with advanced V100 GPUs are less crowded (Figure 20b), with the average allocation ratios of CPUs and GPUs being 35% and 49%, respectively. These machines are usually reserved for a small number of important high-GPU tasks, thus suffering from low utilization. We also observe imbalanced load among V100 machines. In

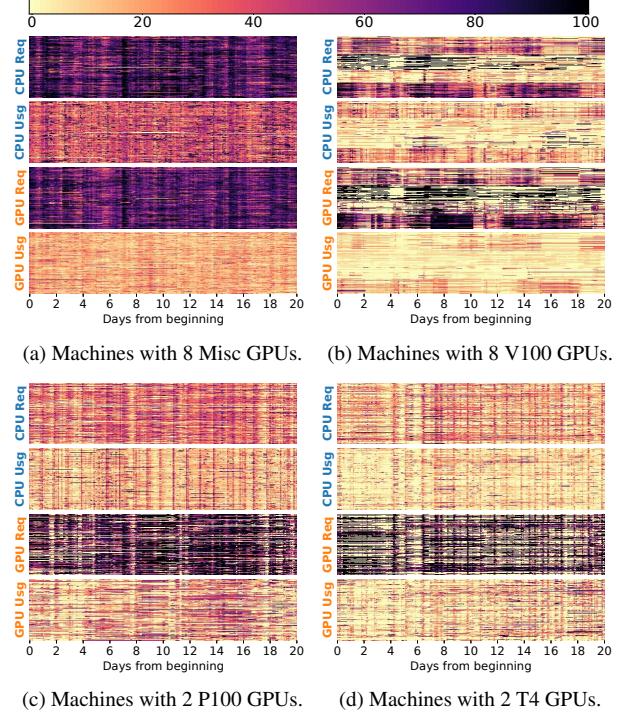
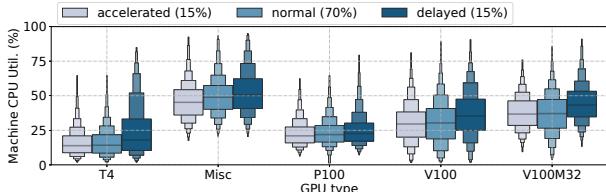


Figure 20: Heatmap of requests and usages of CPU and GPU in machines with different specs. Each row corresponds to one machine.

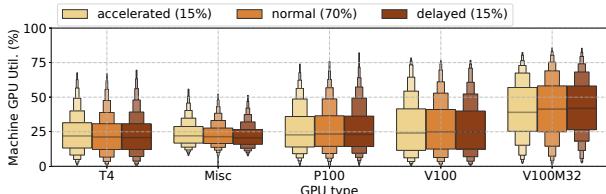
Figure 20b, the machines near the bottom are more crowded than the others. This suggests that the current load-balancing algorithm still has plenty room to improve (Section 6.3).

CPU can be the bottleneck. As shown in Section 6.2, a large number of ML tasks use CPUs more extensively than GPUs. These tasks are more likely to get slowdown in machines with high CPU contentions. To see this, we study the correlation between machine utilization and instance slowdown in the trace and depict the results in Figure 21. Our analysis focuses on the recurring tasks (Section 5.2). In each task recurrence, we divide the instances into three groups: 1) instances with *accelerated* execution whose duration is the shortest 15%, 2) *normal* execution whose duration is the middle 70%, and 3) *delayed* execution whose duration is the longest 15%. Figure 21a compares the CPU utilization in machines running accelerated, normal, and delayed instances. In general, machines running delayed instances measure higher CPU utilization than those running accelerated and normal instances. However, such correlation is not found on GPUs. As illustrated in Figure 21b, the distributions of GPU utilization show no substantial differences across machines running accelerated, normal, and delayed instances.

We next zoom in to the popular CTR prediction tasks with high CPU usage (Section 6.2). Figure 22 shows the CDF of CPU/GPU utilization in machines running accelerated and delayed instances, respectively. In machines with over 24% CPU utilization run 50% delayed instances but only 10%



(a) CPU utilization of machines with various GPU types.



(b) GPU utilization of machines with various GPU types.

Figure 21: Correlation between machine utilization (CPU and GPU) and instance slowdown. Machines hosting delayed instances have higher CPU utilization than those hosting normal and accelerated ones. In contrast, such correlation is not found on GPUs. The boxes depict the $1/128, 1/64, \dots, 1/4, 1/2, 3/4, \dots, 63/64, 127/128$ quantile values [34, 61].

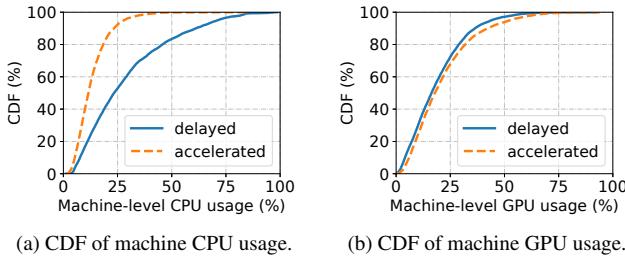


Figure 22: The impact of resource utilization to the execution of CTR prediction instances with high-CPU usages.

accelerated instances (Figure 22a), an evidence of strong correlation between CPU contention and instance slowdown. GPU contention, on the other hand, has no clear contribution to instance slowdown (Figure 22b).

To summarize, task instance scheduling in GPU clusters should also account for the potential interference caused by CPU contentions. This essentially calls for a multi-resource scheduler that jointly considers CPUs, GPUs, memory, I/O, and network when making scheduling decisions.

7 Discussion

Support of elastic scheduling. One fundamental challenge posed to GPU schedulers in heterogeneous clusters is the gang-scheduling requirement of distributed training. Some frameworks [6, 13] are hence developed to support elastic scheduling which allows a training job to dynamically adjust the number of workers on the fly. Compared to gang-

scheduling jobs, elastic-scheduling jobs are easier to handle: they can start with a small amount of resources and later scale to more GPUs when the cluster becomes less crowded. However, elastic scheduling introduces non-determinism to final model accuracy [27, 63].

Machine provisioning and resource disaggregation. GPU schedulers should also account for machine provisioning: in our previous analysis, although 8-GPU machines provide abundant GPU processing power, 2-GPU machines can be a better fit to tasks with heavy CPU processing. To make the problem simplified, many system works propose to decompose monolithic machines into a number of distributed, disaggregated hardware components for improved hardware elasticity [53], despite the non-negligible communication overhead. TensorFlow has recently made a framework-level attempt towards this direction. It released an experimental data service [5] to decouple data pre-processing from GPU training so as to address the CPU bottleneck. However, it requires changing user’s source code with non-trivial efforts.

8 Related Work

GPU sharing. GPU sharing can be supported at different levels. At the GPU hardware level, NVIDIA recently released the Multi-Instance GPU (MIG) [10] feature that enables partitioning a large GPU into multiple small GPU instances with isolated memory and bandwidth. However, MIG is only available on the latest A100 GPUs, and it does not support arbitrary GPU partition. At the GPU software level, GPU time-multiplexing can be implemented by intercepting CUDA APIs [24, 28, 54]. Yet, it usually introduces non-trivial context switching overhead and does not provide a good isolation between the co-located task instances. NVIDIA Multi-Process Service (MPS) [11] offers an alternative solution, but it cannot isolate failures among co-executed process. At the framework level, by extending standard ML frameworks such as TensorFlow and PyTorch, AntMan [66] and Salus [68] enable fine-grained GPU sharing and manage GPU memory for each task instance at a low cost. However, Salus requires users to adapt their code to the framework, while AntMan only supports training tasks.

GPU cluster scheduler. Many GPU cluster schedulers have been proposed recently (Table 1). Notably, Optimus [49] and Tiresias [29] schedule distributed training jobs with an objective of minimizing the average completion time; Themis [41], Gandiva_{fair} [18], and HiveD [72] further consider completion-time fairness for the training jobs. All these works support no GPU sharing, with the minimum allocation unit being one GPU. The clusters used in evaluation are of limited size, workload diversity, and machine heterogeneity.

ML workload characterization. In addition to computation, communication and I/O are also important for distributed training and are thus the focus in the previous characterization

studies. For example, `tf.data` [45] reports that a majority of production ML workloads read many terabytes of data and spend a large proportion of time in data loading. Some ML schedulers [50, 52] study the training efficiency with different network bandwidth and propose to mitigate the communication overhead for accelerated training. An earlier characterization of ML training tasks in Alibaba PAI [59] suggests to replace the PS-Worker architecture with Ring AllReduce to better exploit the high-speed NVLink among GPUs. These works mainly focus on distributed training but leave aside the general MLaaS workloads and cluster resource management.

9 Conclusion

In this paper, we characterized a two-month production trace consisting of a mix of training and inference tasks in a large GPU cluster of Alibaba PAI. We made a number of observations. Notably, the majority of tasks have gang-scheduled instances and are executed recurrently. Most of them are small, requesting less than one GPU per instance, whereas a small number of business-critical tasks demand high-end GPUs interconnected by NVLinks in one machine. For those low-GPU tasks, CPU is often the bottleneck, which is used for data pre-processing and simulation. To better schedule the PAI workloads, our scheduler enables GPU sharing and employs a reserving-and-packing policy that differentiates the high-GPU tasks from the low-GPU ones. We also identified a few challenges that remain open to address, including load imbalance in heterogeneous machines and the potential CPU bottleneck. We have released the trace to facilitate future research on improved GPU scheduling.

10 Acknowledgment

We are deeply indebted to our shepherd John Wilkes, who has patiently gone through this work and helped shape the final version. We thank the anonymous reviewers of NSDI ’22 for their valuable comments. We also thank colleagues from Alibaba Group, including Kingsum Chow, Yu Chen, Jianmei Guo, Guoyao Xu, Shiru Ren, Haiyang Ding, and many others, for their feedback and assistance in the early stage of this work. This work was supported in part by RGC GRF Grant 16213120 and the Alibaba Research Internship Program. Qizhen Weng was supported in part by the Hong Kong PhD Fellowship Scheme.

References

- [1] Alibaba cluster trace program. <https://github.com/alibaba/clusterdata>, 2021.
- [2] Alibaba machine learning platform for AI. <https://www.alibabacloud.com/product/machine-learning>, 2021.
- [3] Amazon machine learning. <https://docs.aws.amazon.com/machine-learning>, 2021.
- [4] Azure AI. <https://azure.microsoft.com/en-us/overview/ai-platform/>, 2021.
- [5] Distributed `tf.data` service. <https://github.com/tensorflow/community/blob/master/rfcs/20200113-tf-data-service.md>, 2021.
- [6] ElasticDL: A Kubernetes-native deep learning framework. <https://github.com/sql-machine-learning/elasticdl>, 2021.
- [7] Google Cloud Vertex AI. <https://cloud.google.com/vertex-ai>, 2021.
- [8] IBM Watson. <https://www.ibm.com/watson>, 2021.
- [9] NVIDIA Management Library (NVML). <https://developer.nvidia.com/nvidia-management-library-nvml>, 2021.
- [10] NVIDIA Multi-Instance GPU (MIG) user guide. <https://docs.nvidia.com/datacenter/tesla/mig-user-guide/>, 2021.
- [11] NVIDIA Multi-Process Service (MPS). https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2021.
- [12] NVIDIA NVLink and NVSwitch. <https://www.nvidia.com/en-us/data-center/nvlink/>, 2021.
- [13] TorchElastic. <https://pytorch.org/elastic>, 2021.
- [14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proc. USENIX OSDI*, 2016.
- [15] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Małgorzata Steinder. Topology-aware GPU scheduling for learning workloads in cloud environments. In *Proc. ACM/IEEE SC*, 2017.
- [16] George Amvrosiadis, Jun Woo Park, Gregory R Ganger, Garth A Gibson, Elisabeth Baseman, and Nathan De-Bardeleben. On the diversity of cluster workloads and its impact on research results. In *Proc. USENIX ATC*, 2018.
- [17] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.
- [18] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proc. ACM EuroSys*, 2020.

- [19] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: QoS-aware resource partitioning for multiple interactive services. In *Proc. ASPLOS*, 2019.
- [20] Maxwell Collard. TensorFlow performance bottleneck on IteratorGetNext. <https://stackoverflow.com/q/48715062>, 2021.
- [21] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proc. ASPLOS*, 2013.
- [22] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proc. IEEE CVPR*, 2009.
- [23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [24] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proc. HPCS*, 2010.
- [25] Raj Dutt, Torkel Ödegaard, and Anthony Woods. Grafana: The open observability platform. <https://grafana.com/>, 2021.
- [26] Yihui Feng, Zhi Liu, Yunjian Zhao, Tatiana Jin, Yidi Wu, Yang Zhang, James Cheng, Chao Li, and Tao Guan. Scaling large production clusters with partitioned synchronization. In *Proc. USENIX ATC*, 2021.
- [27] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.
- [28] Jing Gu, Shengbo Song, Ying Li, and Hanmei Luo. GaiaGPU: sharing GPUs in container clouds. In *ISPA/IUCC/BDCloud/SocialCom/SustainCom*, 2018.
- [29] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *Proc. USENIX NSDI*, 2019.
- [30] Huirong Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. DeepFM: a factorization-machine based neural network for CTR prediction. *arXiv preprint arXiv:1703.04247*, 2017.
- [31] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.
- [32] Tomoki Hayashi, Ryuichi Yamamoto, Katsuki Inoue, Takenori Yoshimura, Shinji Watanabe, Tomoki Toda, Kazuya Takeda, Yu Zhang, and Xu Tan. Espnet-TTS: Unified, reproducible, and integratable open source end-to-end text-to-speech toolkit. In *Proc. IEEE ICASSP*, 2020.
- [33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. IEEE CVPR*, pages 770–778, 2016.
- [34] Heike Hofmann, Hadley Wickham, and Karen Kafadar. Value plots: Boxplots for large data. *J. Comput. Graph. Stat.*, 26(3):469–477, 2017.
- [35] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018.
- [36] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *Proc. USENIX ATC*, 2019. <https://github.com/msr-fiddle/philly-traces>.
- [37] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.
- [38] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. Ray RLLib: A composable and scalable reinforcement learning library. *arXiv preprint arXiv:1712.09381*, 2017.
- [39] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.
- [40] Chi Keung Luk, Jose Americo Baiocchi Paredes, Russell Power, and Mehmet Deveci. Debugging correctness issues in training machine learning models, November 12 2020. US Patent App. 16/403,884.
- [41] Kshitij Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *Proc. USENIX NSDI*, 2020.
- [42] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technol. J.*, 6(1), 2002.

- [43] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proc. ICML*, 2016.
- [44] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.
- [45] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf. data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127*, 2021.
- [46] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *Proc. USENIX OSDI*, 2020.
- [47] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep learning inference in Facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.
- [48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Proc. NeurIPS*, 2019.
- [49] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proc. ACM EuroSys*, 2018.
- [50] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proc. ACM SOSP*, 2019.
- [51] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. MLPerf inference benchmark. In *Proc. ACM/IEEE ISCA*, 2020.
- [52] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *Proc. USENIX NSDI*, 2021.
- [53] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiyi Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proc. USENIX OSDI*, 2018.
- [54] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.*, 61(6):804–816, 2011.
- [55] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proc. IEEE CVPR*, 2016.
- [56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. NIPS*, 2017.
- [57] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.
- [58] Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. Metis: learning to schedule long-running applications in shared container clusters at scale. In *Proc. ACM/IEEE SC*, 2020.
- [59] Mengdi Wang, Chen Meng, Guoping Long, Chuan Wu, Jun Yang, Wei Lin, and Yangqing Jia. Characterizing deep learning training workloads on Alibaba-PAI. In *Proc. IEEE IISWC*, 2019.
- [60] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & cross network for ad click predictions. In *Proc. ACM ADKDD*, 2017.
- [61] Michael L Waskom. Seaborn: statistical data visualization. *J. Open Source Softw.*, 6(60):3021, 2021.
- [62] Shinji Watanabe, Takaaki Hori, Shigeki Karita, Tomoki Hayashi, Jiro Nishitoba, Yuya Unno, Nelson Enrique Yalta Soplin, Jahn Heymann, Matthew Wiesner, Nanxin Chen, Adithya Renduchintala, and Tsubasa Ochiai. ESPnet: End-to-end speech processing toolkit. In *Proc. INTERSPEECH*, 2018.
- [63] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proc. ACM SoCC*, 2016.
- [64] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

- [65] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *Proc. USENIX OSDI*, 2018.
- [66] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU cluster for deep learning. In *Proc. USENIX OSDI*, 2020.
- [67] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. XLNet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.
- [68] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. In *Proc. MLSys*, 2020.
- [69] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proc. USENIX ATC*, 2019.
- [70] Wei Zhang, Wei Wei, Lingjie Xu, Lingling Jin, and Cheng Li. AI Matrix: A deep learning benchmark for Alibaba data centers. *arXiv preprint arXiv:1909.10562*, 2019.
- [71] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proc. VLDB Endowment*, 2014.
- [72] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis CM Lau, Yuqi Wang, Yifan Xiong, et al. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *Proc. USENIX OSDI*, 2020.
- [73] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. Deep interest evolution network for click-through rate prediction. In *Proc. AAAI*, 2019.
- [74] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proc. ACM KDD*, 2018.
- [75] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. AliGraph: a comprehensive graph neural network platform. In *Proc. VLDB Endowment*, 2019.

Evolvable Network Telemetry at Facebook

Yang Zhou[†] Ying Zhang[‡] Minlan Yu[†] Guangyu Wang[‡] Dexter Cao[‡] Eric Sung[‡] Starsky Wong[‡]
[†]*Harvard University* [‡]*Facebook*

Abstract

Network telemetry is essential for service availability and performance in large-scale production environments. While there is recent advent in novel measurement primitives and algorithms for network telemetry, a challenge that is not well studied is *Change*. Facebook runs fast-evolving networks to adapt to varying application requirements. Changes occur not only in the data collection and processing stages but also when interpreted and consumed by applications. In this paper, we present PCAT, a production change-aware telemetry system that handles changes in fast-evolving networks. We propose to use a change cube abstraction to systematically track changes, and an intent-based layering design to confine and track changes. By sharing our experiences with PCAT, we bring a new aspect to the monitoring research area: improving the adaptivity and evolvability of network telemetry.

1 Introduction

Network telemetry is an integral component in modern, large-scale network management software suites. It provides visibility to fuel all other applications for operation and control. At Facebook, we built a telemetry system that has been the cornerstone for continuous monitoring of our production networks over a decade. It collects device-level data and events from hundreds of thousands of heterogeneous devices, millions of device interfaces, and billions of counters, covering IP and optical equipments in datacenter, backbone and edge networks. In addition to data retrieval, our telemetry system performs device-level and network-wide processing that generates time-series data streams and derives real-time states. The system serves a wide range of applications such as alerting, failure troubleshooting, configuration verification, traffic engineering, performance diagnosis, and asset tracking.

While our telemetry system can adopt algorithm and system proposals from the research community (e.g., [18, 27, 48, 50]), a remaining open challenge is *Change*. Changes happen frequently in our network hardware and software to meet the soaring application demands and traffic growth [16]. These changes have a significant impact on the network telemetry system. First, we have to collect data on increasingly heterogeneous devices. This is exaggerated as we introduce in-house built FBOSS [13], which allows switches to update as frequently as software. Second, we have growing applications (e.g., [1]) that rely on real-time, comprehensive, and accurate data from network telemetry systems. These applications introduce diverse and changing requirements for the telemetry system on the types of data they need, data collection

frequency, and the reliability and performance of collection methods.

The changes this paper considers include not only the network events from the monitored data, but also those updates to the telemetry system itself: modification to monitoring intent, advance of device APIs, adjustment of frequency configurations, mutation of processing, and restructure of storage formats. Without explicitly tracking them in our network telemetry system, we struggle to mitigate their impact to network reliability. For example, a switch vendor may change a packet counter format when it upgrades a switch version without notifying Facebook operators. This format change implicitly affects many counters in our telemetry database (e.g., aggregated packet counters), leading to adverse impact to downstream alerting systems and traffic engineering decisions. This example highlights several challenges: (1) Production telemetry is a complex system with many components (e.g., data collection, normalization, aggregation) from many teams (e.g., vendors, data processing team, database team, application teams). A change at one component can lead to many changes or even errors at other components. As a result, when telemetry data changes, it is difficult to discern legitimate data changes from semantic changes. (2) Sometimes, we only detect the error passively when traffic engineering team notices congestion. Yet, we cannot diagnose it easily because the error involves many data. Even worse, it may only affect a small portion of vendor devices due to phased updates. Section 2 shares more such examples.

In this paper, we propose to treat changes as first-class citizens by introducing PCAT, a Production *Change-Aware Telemetry* system. PCAT includes three key designs:

First, inspired by the database community [8], we introduce the *change cube* abstraction for telemetry to explicitly track the time, entities, property, and components for each change, and a set of primitives to explore changes systematically. Using change cubes and their primitives, we conduct the first comprehensive study on change characterization in a production telemetry system (Section 3). Our results uncover the magnitudes and the diversity of changes in production, which can be used for future telemetry and reliability research.

Second, we re-architect our telemetry system to be change-aware and evolvable. In the first version of our telemetry system, we have to modify configurations and code at many devices every time a vendor changes the counter semantics or collection methods, or an application changes monitoring intents. To constrain the impact of changes, i.e., the number of affected components, PCAT includes an intent-based lay-

ering design (Section 4) which separates monitoring intents from data collection and supports change cubes across layers. PCAT enables change attribution by allowing network engineers with rich network domain knowledge to define intents while having software engineers building distributed data collection infrastructure with high reliability and scalability. PCAT then compiles intents to vendor-agnostic intermediate representation (IR) data model, and subsequently to vendor-specific collection models, and job models. The intent-driven layering design reduces the number of cascading changes by 54%-100%, and enables systematically tracking changes through the monitoring process.

Third, we build several change-aware applications that explore the dependencies across change cubes to improve application efficiency and accuracy. For example, Toposyncer is our *topology derivation* service that builds on telemetry data and serves many other applications. We transformed Toposyncer to subscribe to change cubes based on derivation dependencies and greatly reduce topology derivation delay by up to 118s. We leverage correlation dependencies across change cubes to enable troubleshooting and validation.

The main contribution of this paper is to bring the community’s attention to a new aspect of telemetry systems—how to adapt to changes from network devices, configurations, and applications. We also share our experiences of building change-aware telemetry systems and applications that can be useful to other fast-evolving systems.

2 Motivation

To keep up with new application requirements and traffic growth, data center networks are constantly evolving [16]. As a result, changes happen frequently across all the components in telemetry systems, ranging from device-level changes, collection configuration changes, to changes in the applications that consume telemetry data.

Our first generation of production telemetry system was not built to systematically track changes. This brings significant challenges for telemetry data collection at devices, integration of telemetry system components, debugging network incidents, and building efficient applications. In this section, we share our experiences of dealing with changes in our telemetry system and discuss the system design and operational challenges for tracking changes.

2.1 Bringing changes to first-class citizens

We motivate the needs of treating changes as first-class citizen in network telemetry with a few examples.

1. Build trustful telemetry data. Many management applications rely on telemetry data to make decisions. However, in production, telemetry data is always erroneous, incomplete, or inconsistent due to frequent changes of devices and configurations. Moreover, there are constant failures in large-scale networks (e.g., network connection issues, device overload, message loss, system instability). Therefore, applications need

to know which time range and data source are trustful and how to interpret and use the data. This requires tracking changes for each telemetry data value and semantics.

For example, we collect device counters at various scopes (e.g., interfaces, queues, linecards, devices, circuits, clusters). These counters may have different semantics with device hardware and software upgrades or network re-configurations. For example, we have a counter for 90th percentile CPU usage within a time window of a switch. When we change the switch architecture to multiple subswitches [13], we set the counter as the average of 90th percentile CPU of subswitches. However, our alert on this counter cannot catch single subswitch CPU spikes that caused bursty packet drops. We need to know when to change the alerts based on counter changes.

2. Track API changes across telemetry components. Our telemetry system consists of multiple data processing components, which are independently developed by different vendors and teams. When one component changes its interfaces, many other components may get affected without notice. There are no principled ways to handle such changes across telemetry components. For example, vendor-proprietary monitoring interfaces often get changed without an explicit notification or detailed specification. This is because telemetry interfaces are traditionally viewed as secondary compared to other major features. However today cloud providers heavily rely on telemetry data for decisions in a fine-grained and continuous manner. If we do not update data processing logic based on device-level changes, the inconsistency may cause bugs and monitoring service exceptions.

In one incident, a routing controller had a problem of unbalanced traffic distribution, caused by incomplete input topology: a number of circuits were missing from the derived topology. This took the routing team and the topology team over three days to diagnose. The root cause was an earlier switch software upgrade that changed the linecard version from integer (e.g., 3) to string (e.g., 3.0.0). Such a simple format change was not compatible with the post-processing code that aggregated the linecard information into a topology. Thus, we missed several linecards in the topology, which then mislead TE decision and cause congestion in the network. This is not a one-off case, given many vendors and software versions coexist in our continuously evolving networks.

3. Debug with change-aware data correlation. As telemetry components keep evolving, it is hard to attribute a problem to a change using data correlation without explicitly tracking changes and their impacts. For example, when we fail to get a counter, the problem can come from data collection at the device, the network transfer, or both.

In production, we make changes in small phases: first canary on a few devices serving non-critical applications, then gradually on more devices to minimize disruptions to the network [13]. In one incident, there were a small number of devices with “empty data” errors for a power counter. The errors increased gradually and ultimately went beyond 1% threshold

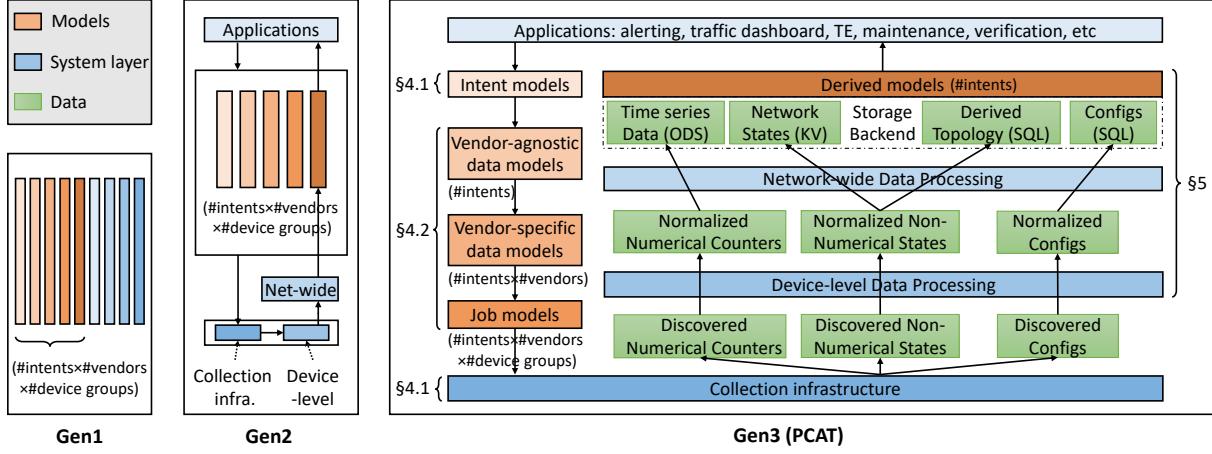


Figure 1: Generations towards change-aware telemetry.

after two weeks and triggered an alarm. This problem was difficult to troubleshoot due to its small percentage. We manually explored the changes through correlation: checking whether there were code changes before the failure, whether the failed devices shared a common region (indicating regional failure), a common vendor, or on common data types. We tried many dimensions of correlation and finally found the errors were mostly related to power and environment counters. The root cause was a vendor changing its format but the processing code could not recognize it. This example shows a tedious manual process of data correlation to debug problems because of gradual change rollouts. To improve debugging, we need to use changes to guide data correlation.

2.2 Lessons from Previous Generations

We now discuss our previous two generations of telemetry systems prior to PCAT and their limitations in handling changes.

Gen1: Monolithic collection script. In a nutshell, a telemetry system is a piece of code that collects data using APIs from the devices. Our first generation is naturally a giant script that codifies what counters to collect. It hardcodes the collection method, polling frequency, post-processing logic, and where to store the data. Figure 1 illustrates Gen1 as intertwined models and system blocks. It runs as multiple cron jobs, each collecting data from different groups of devices. This design is intuitive to implement but is not change-friendly. If a vendor changes the format of a counter, we need to sweep through the entire script to change the processing logic accordingly, and redeploy the new code to all monitors. It has high maintenance burdens as it relies on expert's deep understanding of the code to make changes. Further, tracking changes relies on version control system in the form of code differences, which do not reveal the intent directly.

Gen2: Decoupled counter definition from collection process. As our network expanded, the hulking script in Gen1 became hard to manage. We moved to Gen2, which separates the monitoring model (i.e., what counter to collect) from the actual collection code, shown as orange and blue boxes in Fig-

ure 1. The separation allows us to track changes to data types separately from the collection logic (e.g., sending requests, handling connections). However, the intent is still mingled with the vendor-specific counter definition. For instance, one may want to collect the “packet drops per interface”. One needs to specify the exact SNMP MIB entry name and the specific API command. A low-level format change would result in updates on all model definitions. Moreover, the data collection system includes both the collection infrastructure and data processing logic. The data processing logics scatter across many places, e.g., when the data is collected locally at the collector, or before it is put into the storage. To change a piece of processing logic, we have to change many such places, which is cumbersome to track. In addition, when a piece of data is changed or is absent, tracing back on what causes the change is manual and tedious.

2.3 Challenges and PCAT Overview

Our experiences of previous two generations indicate three main challenges in handling changes: change abstraction, attribution, and exploration. To address these challenges, we build our Gen3 telemetry system – PCAT.

Change abstraction. In Gen1 and Gen2, changes were not stored structurally. They exist either as diffs in code reviews in Gen1, or logs to temporary files in Gen2. Without a uniform representation, each application needs to develop ad-hoc scripts to parse each data source. This leads to not only duplicate efforts but also missing changes or mis-interpretations. A uniform and generic change abstraction allows hundreds of engineers to publish and subscribe to changes to boost reliable collaboration without massive coordination overhead. In §3, we propose a generic abstraction called *change cube* to tackle this challenge.

Change attribution. The second challenge is the turmoil to ascribe the intent of the scattering changes. The solution involves a surgically architectural change to a multi-layer design, shown in Figure 1 and elaborated below.

Data collection. The first step is to collect data from de-

vices, called *discovered data*. There are three types: numeric counters, non-numeric states, and configurations (see Table 4 in Appendix). We use different protocols for collecting different data and for different devices: SNMP [10], XML, CLI, Syslog [28], and Thrift for our in-house switches [13].

Device-level data processing (normalization). The data is different in formats and semantics across devices, vendors, and switch software. This makes it difficult for applications to parse and aggregate the data from different devices. We use a device-level data processing layer to parse the raw data to a unified format across devices, vendors, and switch software.

Network-wide data processing. Next, we aggregate device-level (normalized) counters, states, and configurations into network-wide storage systems for applications to query. The normalized non-numerical states (as network states) are stored in a key-value store. We build a tool called **Toposyncer** which constructs *derived topology* from normalized non-numerical states. For example, from per-device data, we can construct the device, its chassis, linecard, as well as cross-device links.

Data consumer applications. There are many critical network applications that consume PCAT data. Network health monitoring and failure detection use monitoring data to detect and react to faults. Network control relies on real-time data for making routing and load balancing decisions [2, 38]. Maintenance and verification use telemetry data to compare network states before and after any network operations.

There are several advantages of the new design compared to previous generations. First, compared to Gen2, Gen3 dissects a monolithic data definition into three different types, each focusing on defining one aspect of the monitoring. The separation brings better scalability and manageability. We describe the details in §4. Second, we not only care about tracking changes in data format and code, but also need to attribute changes to the right teams (i.e., who/what authored the change). Change attribution builds the trust of the data for applications. It facilitates collaboration across teams towards transparent and verifiable system development. Gen3’s intent-based layering design lets each team play by their strength and work together seamlessly. Specifically, the network engineers can leverage their rich domain knowledge and focus on intent definition, while software engineers focus on scaling the distributed collection system.

Change exploration. Many designs and operations require a clear understanding of the relations amongst changes. For example, to debug why a piece of data is missing, we always find the last time the data appears and check what has changed since then. We may find one change to be the cause, which could be caused by another change somewhere else. Similarly, when receiving a change of an interface state, we need to reflect the change on the derived topology and upper-layer applications. It motivates us to develop primitives for change exploration that serves many applications. We demonstrate the usage in real-time topology derivation in §5.

3 Changes in Facebook Network Telemetry

In this section, we define the change cube concept and explain how they are generated in this system, together with extensive measurement results by composing queries on top of the change cubes.

3.1 Change Cube Definition

To systematically handle changes in network telemetry, we leverage the concept of *change cubes*. Change cubes are used in databases [8] to tackle the data change exploration problem by efficiently identifying, quantifying, and summarizing changes in data values, aggregation, and schemas. Change cube defines a set of schemas for changes and provides a set of query primitives. However, changes in network telemetry are different from those in databases in two aspects: (1) Network telemetry generates streaming data with constant value changes, so the change cubes in network telemetry do not care about value changes but only changes in schema and data aggregation. (2) Network telemetry has frequent changes due to fast advances of hardware and software that result in data semantics changes.

Change cubes. We define a change cube to be a tuple $\langle Time, Entity, Property, Type, Dependency \rangle$. We summarize each field of the change cube in Table 1 and explain below.

- *Time* dimension captures when the change happens. It depends on the granularity we detect changes, e.g., seconds, minutes, or days.
- *Entity* represents a measurement object, e.g. a switch, a linecard, as well as the models that describe what to measure and how.
- *Property* contains the fields or attributes of the entity that get changed. For example, a loopback IP address of a switch, an ingress packet drop of an interface.
- *Layer* dictates the layer or component in the telemetry system (in Figure 1) where changes happen. We discuss how we land in these choices in §4.
- *Dependency* dimension contains a list of other changes that this change is correlated with. Each item in the list is a $\langle ChangeCube, Dependency Type \rangle$ pair. We support two dependency types: correlation dependency and derivation dependency. Derivation dependency means that a lower-layer change causes an upper-layer change. Correlation dependency means two changes on correlated entities or properties.

Primitives on change cubes. Next, we introduce the operators on the change cube, which are used to explore the change sets. We leverage the operators proposed in [8] but redefine and expand them in the context of telemetry systems.

- $Sort_f(C)$ applies function f to a set of change cubes C , on one or a few dimensions to a comparable value, and uses it to generate an ordered list of C . In our problem, sort is mainly used with time to focus on the most recent changes.
- $Slice_p(C)$ means selecting a subset of C where the predicate

Dimension	Sub category	Examples
Time	Multiple time granularities	Second, minute, hour, day
Entity	Intent model	High-level intent, e.g., packet drops at spine switches
	Vendor-agnostic data model	Counter scope, unit
	Vendor-specific data model	Format, API
	Job model	Collection channel, frequency, protocol
	Derived model	Derived network switch
Property	Model fields	IP address, network type
	Change attributes	LoC, reason
Layer	Application	Adding alert to detect a new failure type
	Network-wide processing	Topology discovery code logic
	Device-level processing	Normalization rule
	Collection infrastructure	Codebase for collection tasks
Dependency	Correlation dependency	BGP session and interface status
	Derivation dependency	Circuit is derived from two interfaces' data

Table 1: Change Cube Definition.

p is true. It is used to filter an entity or a property value.

- $\text{Split}_a(C)$ partitions C to multiple subsets by attribute a . An example is to split the changes by the layer to group changes according to where they occur. A reverse operator to Split is Union , which combines multiple change sets.
- $\text{Rank}_f(P_C)$ After we split C to multiple sets P_C , we further analyze these sets and rank them based on a function, e.g., cube size, the time span, the volatility.
- $\text{TraceUp}(c)$ and $\text{TraceDown}(c)$. These two operators are used with the *Dependency* field, which are new compared to [8]. The former traces the changes that the current change c depends on, and the latter traces the changes that depend on the c . They are useful for debugging through layers and validation across data.

Explicitly tracking changes in a structured representation eases the diagnosis process. Considering the second example in §2.1, when the switch software is updated, it populates a change cube to the database, indicating the API's return result has changed. Consequently, it triggers another change cube at the counter model level on this specific CPU counter. This change cube in turn propagates through the monitoring stack to job changes and retrieved data changes. The applications using the CPU counters can subscribe to such data change, which can then be notified immediately. The chain of change notifications eliminates the post-mortem debugging after the counter change causes application errors.

3.2 Changes in PCAT

Leveraging *change cubes*, we provide the first systematic study of changes and their impact on network telemetry systems. We populate change cubes of PCAT using multiple ways. For the data stored in database, we leverage our database change pub/sub infrastructure [39]. We subscribe to the telemetry objects' change log and translate them to change cubes. For code changes in collection infrastructure, data processing logics (both device-level and network-wide), and

Queries	Formulas
Q1 (Fig. 2a)	$\text{Sort}_{\text{Time(Week)}}(\text{Slice}_{\text{layer}}=\text{"application"}(C))$
Q2 (Fig. 2a)	$\text{Sort}_{\text{Time(Week)}}(\text{Slice}_{\text{entity}}=\text{"vendor-agnostic data model"}(C))$
Q3 (Fig. 2c)	$\sum_c c.\text{LoC}, c \in \text{Split}_{\text{Time(Week)}}(\text{Slice}_{\text{layer}}=\text{"application"}(C))$
Q4 (Fig. 3a)	$\text{Sort}_{\text{Time(Day)}}(\text{Slice}_{\text{entity}}=\text{"job model"} \& \text{property}=\text{"frequency"}(C))$
Q5 (Fig. 3b)	$\text{Split}_{\text{network type}}(\text{Slice}_{\text{entity}}=\text{"job model"} \& \text{property}=\text{"frequency"}(C))$
Q6 (Fig. 3c)	$\text{Split}_{\text{network type}}(\text{Slice}_{\text{entity}}=\text{"job model"} \& \text{property}=\text{"channel"}(C))$
Q7 (Fig. 4a)	$\text{Split}_{\text{blueprint type}}(\text{Slice}_{\text{layer}}=\text{"application"} \& \text{reason}=\text{"blueprint"}(C))$
Q8 (Fig. 4b)	$\text{Split}_{\text{vendor}}(\text{Slice}_{\text{layer}}=\text{"application"} \& \text{reason}=\text{"new model"}(C))$

Table 2: Queries used in §3.2.

applications, we parse the logs in the code version control system to generate change cubes. Intent model, data model (both vendor-agnostic and -specific), and job model changes are codified and thus tracked through code changes [41]. They can be populated using the same way as other code changes. We store all change cubes to a separate database called *ChangeDB* and develop APIs to explore these changes.

We analyze changes from the perspectives of devices, collection configurations, and application intents, over seven years (2012-2019). Our results below uncover surprisingly frequent changes and quantify the diverse causes of changes.

3.2.1 Change Overview

Change frequency. We first quantify the code changes of our monitoring system. We map one code commit to one change cube, involving multiple lines of code across multiple files. We group the changes into three categories according to where they happen in Figure 1: collection infrastructure (bottom layer), data & job models and processing (middle two layers), and applications, representing the infrastructure, data, and intent respectively. We construct queries using the primitives defined earlier. We put the actual query to generate the figures in Table 2. Q1 uses *Slice* to filter the changes in application layer, and sorts the changes by time. We replace the “application” with other values for changes in other layers. Q1

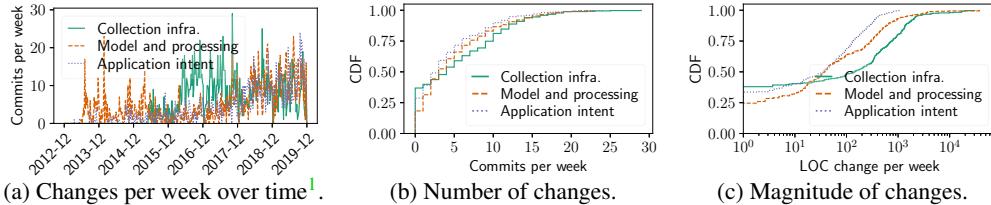


Figure 2: Change characteristics.

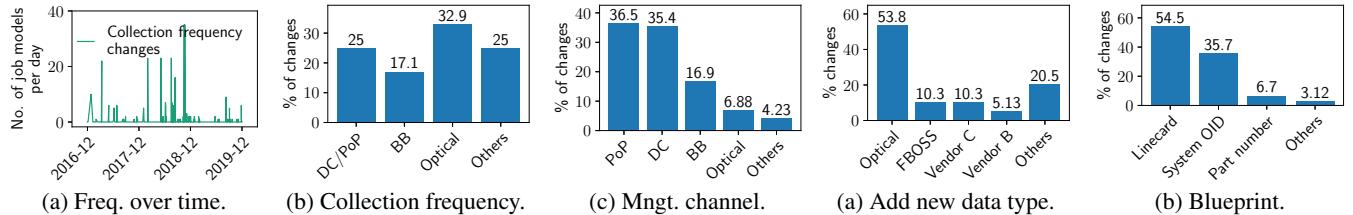


Figure 3: Collection configuration changes².

can be compiled into the following SQL: `SELECT COUNT(*) FROM ChangeDB WHERE layer = "application" GROUP BY time_week ORDER BY time_week.`

Figure 2a shows the number of changes per week. We find that *types of changes vary greatly as the telemetry system scales*. More model and processing changes occur at the beginning (the year of 2013), as we begin by adding more counters to monitor. When the number of counters reaches a certain scale (the year of 2016), we realize the infrastructure needs better scalability. Thus there are more changes to refactor the collection infrastructure. Application intent follows the same trend as data changes, as adding new data is often driven by the need from applications.

Cumulatively across time, we show the average numbers of weekly changes of three categories in Figure 2b. They are on the same order of magnitude, with slightly more infrastructure changes. It can be as high as 25-30 changes per week. Note that each change is deployed on many switches and the changes it introduces to the network is significant.

Change magnitude. We quantify the magnitude of changes in terms of Lines of Code (LoC) using query Q3. While most code changes are not big, some changes could touch multiple lines due to consolidation of processing logic and refactoring. This is obtained by first getting a slice of changes of a given category, splitting the changes into weeks, and summing up the LoC property. Figure 2c shows that *collection infrastructure has larger changes and the application has changes with larger volatility*. We can dig into the volatility of each change set by computing its variance and use the *Rank* primitive. Both figures show there exists a significant number of large changes. For example, there are 27 weeks with more than 1000 LoC changes for collection infrastructure. However, as the industry’s trend is to move away from monolithic changes

¹The collection and processing infrastructure were not merged into the codebase before 2015-04; so its commits are non-trackable before that.

²The “Other” contains some changes that are hard to classify programmatically. The same applies to Figure 4.

Change reasons	%
Collection infrastructure	67.9
Adding new devices	17.8
Topology processing	8.30
Data format	4.86
Counter processing	1.19

Table 3: Change categorization by change reasons.

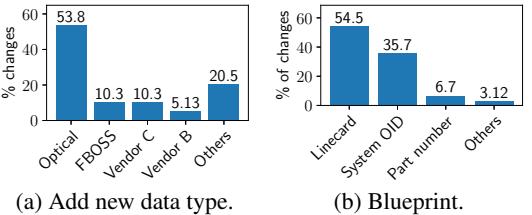


Figure 4: Network configuration changes.

to many small incremental changes [13], we expect to have more frequent small changes going forward.

Change reason categorization. We analyze the breakdown by reason of change using $\text{Split}_{\text{reason}}(C)$, which is obtained by parsing the commit log text and adding it to the ChangeDB. Table 3 shows that one major reason is collection infrastructure changes (67.9%). Adding new devices to the network is the second dominant reason (17.8%). Topology processing changes occupy 8.3%. The fourth reason is adjusting the data formats of collection models (4.86%). Lastly, 1.19% come from the device-level counter processing code.

3.2.2 Device-Level Changes

In our large-scale networks, we constantly add new vendors and devices to leverage a rich set of functions and to minimize the risk of single-vendor failures. The number of devices increased 19.0 times and the number of vendors increased 4.7 times as observed by PCAT in six years. Even with the same vendor, we gradually increase the chassis types, which have different combinations of linecard slots and port speeds. More choices of chassis types allow us to have fine-grained customization to our network needs. Furthermore, the number of chassis types grows from 26 to 129 (4.4×). In addition, our in-house software switch has tens of code changes daily and deploys once every few days [13].

3.2.3 Collection Configuration Changes

Collection frequency. Applications adjust the collection frequency to balance between data freshness and collection overhead. We first analyze collection changes by counting daily changes of collection frequencies over time, using query Q4. Figure 3a shows that there are constant collection frequency changes over time, with more frequent changes near December 2018 – because of tuning collection frequencies for newly-added optical devices. We analyze collection frequency changes by applying *Slice* on both the entity and the property. Interestingly, Figure 3b shows that *optical devices*

change frequency more often (32.9%) because they cannot sustain high-frequency data polling and thus require more careful frequency tuning.

Management channel changes. PCAT collects data from the management interfaces at devices. As our management network evolves, we frequently reconfigure management interfaces (e.g., IP addresses, in-band vs. out-of-band interfaces). Backbone and PoP devices have multiple out-of-band network choices for high failure resiliency. Figure 3c breaks the IP preference changes into PoPs, DCs, and Backbones. *PoPs have more frequent channel changes* (36.5%) because PoPs are in remote locations and thus have more variant network conditions. Selecting the right channel is important to keep the device reachable during network outages so that we can mitigate the impact quickly.

3.2.4 Application Intent Changes

Data type changes. PCAT supports an increasing number of diverse applications over years. Applications may add new types of data to collect (e.g., to debug new types of failures), or remove some outdated data. Figure 4a shows how different vendors add new data types. Optical device vendors add more data (53.8%) because we recently start building our own optical management software and thus need more counter types. Indeed, optical devices generally have more types of low-level telemetry data compared to IP devices, e.g., power levels, signal-to-noise ratio. They are also less uniformed across vendors than IP devices.

Hardware blueprint changes. Hardware blueprint specifies the internal components (chassis, linecards) of each switch and determines what data to collect. Figure 4b shows the percentage of changes for hardware blueprints such as linecard map, system Object Identifier (OID) map, part number map, and others (e.g., OS regex map). These changes are due to network operations such as device retrofit and migration. They may cause data misinterpretation if not treated carefully.

4 Change Tracking in Telemetry System

In this section, we describe the layering design of our current intent-based telemetry system to help track changes.

4.1 Towards change-aware telemetry

Intent modeling. We use a thrift-based modeling language that empowers network engineers to easily specify their monitoring intents. Compared to other intent language proposed in academia [19, 31, 32], our language puts more emphasis on device state in addition to traffic flows, and defines actions in addition to monitoring. Our language contains three components shown in Figure 5.

- *Scope* captures both the device-level scope (e.g., Backbone Router) and network-level scope, (e.g., DC fabric network).
- *Monitor* specifies what to monitor in a vendor-agnostic way. For example, an intent could be capturing packet discard for the gold-level traffic class, which will get translated

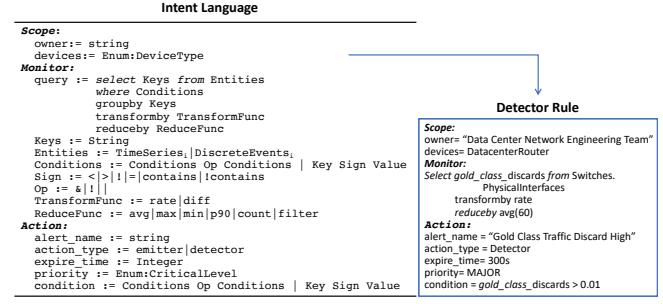


Figure 5: Intent model.

```

ModelDef(
  name='PhysicalInterface',
  properties=[PropertyDef(name='if_name',type=STRING,transform=NONE),
  PropertyDef(name='if_hc_in_octets',type=INT64,transform=RATE),
  PropertyDef(name='if_in_discards',type=INT64,transform=RATE),
  ...],
  children=[ModelDef(
    name='GoldQueueCounters',
    properties=[PropertyDef(name='queue_name',type=STRING),
    PropertyDef(name='packet_count',type=INT64),
    PropertyDef(name='byte_count',type=INT64),
    PropertyDef(name='packet_discards',type=INT64)]),
  ModelDef(
    name='SilverQueueCounters',
    ...),
  ...],
  ...
  )
  
```

Figure 6: Data model.

to a specific SNMP MIB entry or particular counters. In the left part of Figure 5, we describe the SQL-like query language. The *keys* are monitored metrics and the *entities* are time-series data streams and discrete events tables. We also support data aggregation functions such as *avg*, *count*, *filter*, which aggregate samples over time and devices.

- *Action* includes two types: Emitter and Detector. Emitters subscribe to *discrete network events* that are pushed from devices, and define actions upon receiving these events. Detectors allow us to write formulas for various time-series data, and set up a threshold for the formula value as the alerting condition. A detector example is shown in the right part of Figure 5; it defines a detector based on the key *gold_class_discards* which captures the packet drops for gold-class traffic on a physical interface. The discard is transformed to rate, and aggregated every 60 seconds. The alert is triggered if the threshold is greater than 0.01.

The intent model hides low-level changes. Vendors may change the queue drop counter names, or the mapping between queues and gold-class traffic may change. The intent configuration remains unchanged in both cases.

Runtime system. We handle heterogeneous intents with homogeneous software infrastructure. Thanks to separation, software engineers can focus on the runtime execution system to solve the hard system building problems: scheduling, load balancing, scalability, and reliability. The runtime execution system collects data from devices according to the model, which includes a distributed set of engines and a centralized controller to distribute jobs and collect data from these engines. The centralized controller fetches the latest collection and job models, combines with device information in our database, and generates a sequence of jobs to be executed

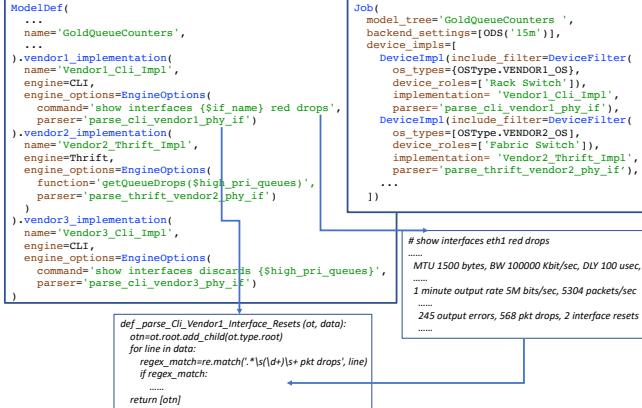


Figure 7: Collection method and job model.

with a given deadline. It dispatches jobs to designated engines based on load and latency. The engine executes the collection command, performs device-level processing, and sends data back to the corresponding storage. The system is heavily engineered to tackle the reliability and scalability challenges.

4.2 Change reduction w/ vendor-agnostic IR

Next, we zoom into the intermediate layer between the intent and the runtime. The high-level monitoring intent is translated to the intermediate representation data model, which gets mapped to the vendor-dependent collection model, and finally is materialized to the job model on each device. We emphasize that how the modular design principle is translated to different models in order to *limit the impact of changes*.

Vendor-agnostic intermediate representation (IR) data model. The data model is created based on the *keys* field in the intent model. It specifies data schema in the following way, as shown in Figure 6.

- *Hierarchical.* We choose a tree structure as an intermediate representation, called the *model tree*. An example is shown in Figure 7. An `AggregateInterface` model has multiple child models, e.g., `PhysicalInterface`, `BGPSessions`. A `PhysicalInterface` also has multiple child models. Models are like templates waiting to be filled in. When they are materialized by actual monitoring data, we call them **objects**. By organizing the materialized objects in the same hierarchy as the model tree and adding a dummy root to connect up the top-level objects, we get a materialized **object tree**. The models define the data to be collected, which is derived from the *keys* field in the intent model.
- *Typed.* The data model defines the types of data to make interpretation of the data easier, e.g., `if_hc_in_octets` is the incoming traffic in octets.
- *Processing instruction.* It also defines basic processing primitives to go with the data using the *transform* field, e.g., computing a per-second rate from consecutive absolute counts. Both the type and the processing instruction are determined by the intent. Placing all the processing logic in a separated blob makes it much easier to track the changes

in processing logic.

Vendor-dependent collection model. The IR model is further compiled down to vendor-specific counter names, specific commands to use, e.g., a CLI command, Thrift function name. Figure 7 shows two collection methods for the `GoldQueueCounters` data model: CLI and thrift. In each implementation, we define the collection API and the post-processing function in the *parser* field. We show an example of the CLI parser function that matches the regex in the output of a command on the vendor1 device. Creating this layer of model separately allows us a place to capture all changes due to vendor format and API changes, which are quite common.

Vendor-dependent job model. The job model combines the collection method with a concrete set of devices, shown in Figure 7. The *implementation* field matches with what is defined in the collection method. Instead of defining a job spec for each device, we group devices and apply the same job spec for all of them. Figure 7 uses `DeviceFilter` to define device role (e.g., rack switches), OS type, region, device state, etc. Job models are the input to the runtime execution to handle job scheduling and manage job completion. Job model captures the system aspects of changes. It can be adapted according to performance and scalability requirement, which is independently controlled from the intent or data specifications.

5 Change Exploration

Once PCAT collects data based on monitoring intents, we run device-level and network-level processing to report the data back to applications. Below, we build a few change-aware applications by exploring dependencies across change cubes.

5.1 Change-driven Topology Derivation

Toposyncer is our topology generation service, part of the collection infrastructure (see network-wide data processing in Figure 1). It creates *derived topology* from normalized device-level data (i.e., in vendor-agnostic format) (Figure 8). For example, from per-device data (e.g., interface counters, BGP sessions), Toposyncer constructs the device, its chassis, linecard, as well as cross-device links.

Toposyncer overview Toposyncer has four processes: (1) `Sync_device` constructs nodes with multiple sub-components: sub-switches, chassis, line cards (line 2-11 in Figure 9). It also derives device-level attributes such as power and temperature, control and management plane settings. (2) `Sync_port` derives physical and logical interfaces on each node and their settings (IP address, speed, QoS) (line 12-13). (3) `Sync_circuit` constructs cross-device circuits. A circuit is modeled as an entity with two endpoints, pointing to the interface of each end's router [41]. For each interface, it searches for all possible neighbors based on various protocol data, e.g., LLDP, MAC table, LACP table. In case some data source is incomplete, we search all data sources, independently identify all possible neighbors from each data source, and consolidate the results.

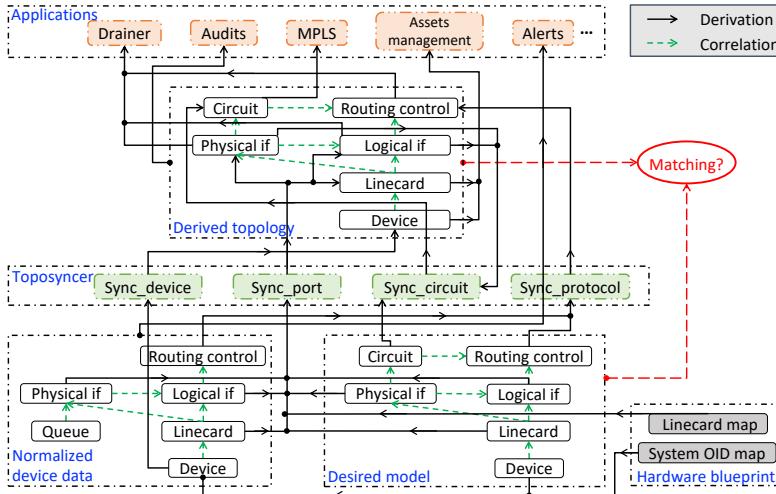


Figure 8: Data dependency graph. Toposyncer consumes the device data, desired model and hardware blueprints to generate derived data. PCAT verifies the derived data with the desired model.

(4) *Sync_protocol* creates the protocol layers on top of the circuits, such as OSPF areas, BGP sessions and their states.

Toposyncer uses two additional data sources as templates to guide the construction: the desired model which defines the operator’s intent topology and hardware blueprints which include hardware specifications, as shown at the bottom in Figure 8. Figure 9 shows the process. It uses desired device data (names, IP addresses) to decide what device to derive (line 2). Then, it uses the hardware blueprints and desired data to handle ambiguity. For instance, to figure out “what is this chassis”, it first checks the discovered chassis name in raw data from the device. But often the discovered name is not uniquely mapped to a chassis but to several possible chassis versions, e.g., two versions with 4 linecards, one version with 8 linecards. Toposyncer cross-checks with hardware blueprints and picks the best match³ (line 6-8). This process is similar to other topology services [29, 41], but we focus on derived models and how we populate them automatically from telemetry data.

Improve Toposyncer with change cubes. Our first implementation of Toposyncer did not utilize changes. It ran periodically against the latest snapshots of collected data at a fixed frequency (e.g., 15 minutes). This method leads to stale derived data, which affects the freshness and accuracy of upper-layer applications. Another challenge is debugging. When a piece of data (e.g., a circuit) is missing in the derived topology, it is hard to find out whether it is because of a raw data change, a normalized data change, a desired model change, a hardware blueprint change, or other reasons. We tackle these problems using change cubes and the dependency primitives below.

³When the guess is wrong, it exhibits as a discrepancy between desired and derived topology. We add alarming to detect such differences and involve humans to manually investigate.

```

1: procedure DERIVETOPOLOGY(Collection, Desired,
   HdwTemps, DependencyG)
2:   for d ∈ Desired.Devices do
3:     DeviceObj, dep = sync_device(Collection, d)
4:     DependencyG.add(dep)
5:     blueprint = getBlueprint(HdwTemps, d)
6:     for chassis_temp ∈ blueprint do
7:       derived_chassis = sync_chassis(Collection,
          chassis_temp)
8:       for linecard_model ∈ chassis_temp do
9:         derived_chassis.add(sync_linecard(Collection,
            linecard_model))
10:        DeviceObj.addchassis(derived_chassis)
11:    DeviceObjs.add(DeviceObj)
12:    for d ∈ DeviceObjs do
13:      derived_ifaces = sync_port(Collection, d)
14:      for iface ∈ derived_ifaces do
15:        neighbors.add(findNeighbors(iface, Collection))
16:        circuits = sync_circuits(iface, neighbors)
17:        sync_protocol(Collection, circuits)
18: procedure UPDATETOPOLOGY(UpdateQ, DependencyG)
19:   while UpdateQ ≠ 0 do
20:     changei = UpdateQ.pop()
21:     dependent_objs = changei.Dependency
22:     update_func=findFunc(dependent_objs)
23:     update_func(dependent_objs, changei)

```

Figure 9: Toposyncer algorithm

Build change cubes. We generate change cubes for normalized data, desired model, hardware blueprint, as well as Toposyncer code changes, shown as each dotted box in Figure 8. We generate these cubes by parsing database transaction logs and model/code changes from version control system logs and publish them to ChangeDB. For example, when an operator changes the configuration of an SNMP MIB for a device, we generate a record to the DB.

Derivation dependency. We populate the derivation dependency across change cubes A and B if we derive data A from data B. In the above example, the MIB change will result in multiple change cubes of job models. We build the dependency between the MIB config change and the rest of job model changes. Figure 8 shows derivation dependency in solid arrows across objects in different layers (each large dash box representing a layer). The dependency exists between data objects as well as between code and data.

Subscription to change cubes. Toposyncer subscribes to the change cubes and invokes corresponding processing logic accordingly, shown in line 19-23. For example, *sync_port* subscribes to the device data (e.g., *Thrift_Fboss_Linecards*, *Snmp_entPhysicalTable*), and a hardware blueprint (i.e., linecard map). If the hardware blueprint changes, i.e., the same linecard name is mapped to a different hardware blueprint, the change cube will be published and *sync_port* triggers its function *sync_phy_iface* function on the impacted interfaces. Similar pub/sub relation is also built between applications and derived data. For instance, as shown in Figure 8, a drainer application subscribes to interface status and the routing control messages to determine if it is safe to perform an interface drain operation.

5.2 Improve Trust on Data Quality

Real-world telemetry data may contain dirty or missing data. By exploring the change history, one can better judge whether the current values are trustworthy. Observing patterns of data changes can help predict the occurrences of future changes and identify missing changes.

Correlation dependency. Amongst normalized device data or derived data, data have relationships between them, shown as dash arrows within each large dash box in Figure 8. The relationship represents the physical dependency across objects, such as “contain”, “connect”, “originate”. Previous topology-modeling works Robotron [41] and MALT [29] focus on the desired model and the correlation dependency in it. The desired model is built for the purpose of capturing topology intents and generating configurations. Here we use the model together with change cubes to verify if the actual topology’s change is legit by comparing it against the desired models. A change cube generated from the desired object should have a matching change cube in the derived object, and vice versa. This can be done with a *Slice* on the entity, *Sort* by time, and compare the *Entity* of the changes.

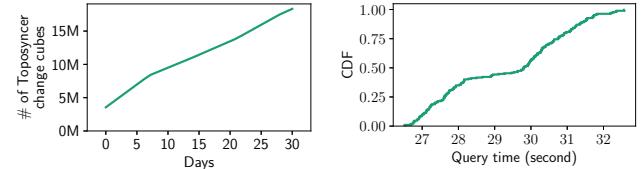
This correlation dependency can also be used for cross-layer validation of data quality. We implement *if-then* validation rules based on the correlation dependency on change cubes. We give two examples below. One use case is hard-stop fault detection. One rule is that *if* the logical interface fails (i.e., a specific change cube on a logical interface), *then* the routing session going through it will also fail (i.e., another change cube on the routing session must exist). If we observe significant errors at the lower layer but no upper failure, it indicates a measurement issue. In another use case, the aggregate interface consists of multiple physical interfaces. *If* a member physical interface reports packet errors, *then* the packet errors from aggregated interface should be larger than or equal to the physical interface errors. If the rule is not satisfied, it indicates some issues. These cross-layer dependencies can help us detect change-induced problems more quickly.

6 Evaluation

This section evaluates how the layer design of PCAT has helped with change tracking and how much benefit the change cube method has brought to use cases.

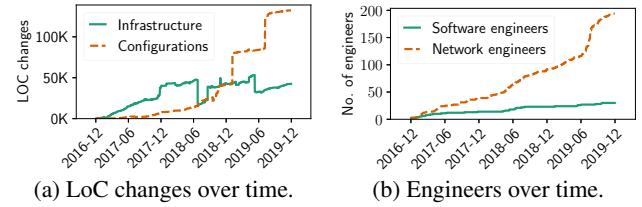
6.1 Change tracking implementation

First, we examine whether tracking all the changes is even feasible in a production environment. We show the change cube data volume grows with time in Figure 10a. Drawing from the experiences of Facebook’s data infrastructure team, we employ a two-tier storage solution. We have an in-memory database to hold the change data for the most recent 30 days and have a disk-based SQL database for longer historical data. At the same time, the change data is published to our publish/subscribe system [4] for real-time propagation. Next, we



(a) # of change cubes over time. (b) Query distribution time.

Figure 10: Scalability and performance of change tracking.



(a) LoC changes over time. (b) Engineers over time.

Figure 11: Separating configurations with telemetry infra.

evaluate the performance of exploration using the primitives defined in §3.1, which is implemented using SQL statements. Figure 10b shows the query distribution time for data stored on disk, most of which centers around 27-32 seconds, due to the large data volume. For shorter duration of data in memory, it takes less than one second.

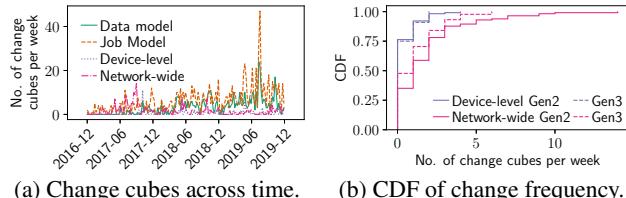
6.2 Benefits of separation

Analyzing change data over time helps us evaluate the long-term benefit of the layer design. We show it from three aspects.

Decoupled evolution of configurations and infrastructure. We categorize changes broadly to configuration changes vs. infrastructure changes. We quantify the magnitude of the change using the Lines of Code (LOC) change. Figure 11a shows that the changes for configurations are 3.1 times more than core collection infrastructure changes. The sudden jumps for configurations in January 2019 are due to adding a large set of optical devices, which was not monitored by PCAT. The second increase around July 2019 is due to the migration to Gen3, resulting in a large number of new models added. The result shows that we increase the monitoring scope by configuration layer changes with a stable infrastructure.

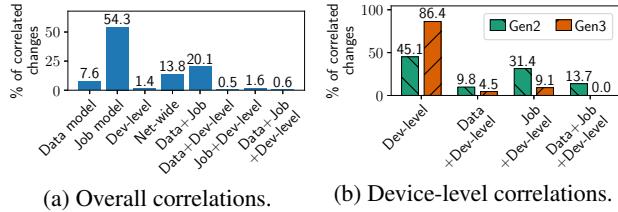
Scaling with divided responsibility. The separation in software systems has a long-term impact on the organization growth and people aspects. In Figure 11b, we analyze the change authors and categorize by their roles. It shows the number of network engineers who have made changes to configurations is increasing at a much faster pace than software engineers, with 7.2 times more people recently. The increase around June 2019 is due to both migration to Gen3 and adding more optical devices to monitor. It is clear that both of these changes are carried out by network engineers. It shows that PCAT enables network engineers to work on different network types while a small number of software engineers maintain infrastructure. It will boost a healthy collaboration environment where each team can play by their strength.

Confining the impact of changes. We use the number of change cubes as an approximate of the volume of changes.



(a) Change cubes across time. (b) CDF of change frequency.

Figure 12: Change cube frequency.



(a) Overall correlations. (b) Device-level correlations.

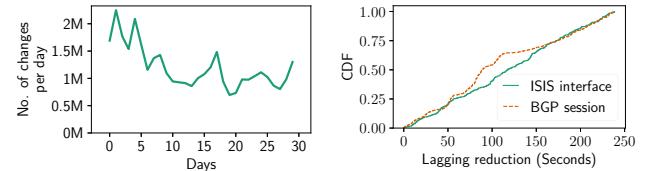
Figure 13: Correlated changes.

Figure 12a shows its trends across time for data models, job models, device-level processing, and network-wide processing. The maximum numbers range from 15 to 50 for different categories. The models (data and job) have more changes due to the frequent intent changes. The infrastructure layers (device and network-wide processing) are more stable. Recently there are more data model and job model changes, because of Gen2-to-Gen3 migration. To directly illustrate the benefit of modular design in Gen3 (§4.2), Figure 12b compares the frequency of change cubes for device-level and network-wide processing in Gen2 and Gen3 (after 2019-02). We observe that the average change frequency for network-wide processing in Gen3 is 38.1% lower than Gen2, while device-level remains similar. This means the modular design in Gen3 further prevents the changes in lower data model and job model layers from impacting upper processing layers, confining the impact of lower-layer changes. Note that we discounted the changes due to Gen2-to-Gen3 migration to have a fair comparison.

Reducing correlated changes. We find change cubes that occur close in time as correlated changes (e.g., data and job models are modified in the same commit). We show that PCAT’s way of separating layers and models has helped reduce correlated changes. We first present the breakdown of different correlation combinations in both generations in Figure 13a. The largest combination is data and job, accounting for 20.1%. It is because adding new devices requires adding both data and job models. There are a small fraction of changes that require updating data, job, and device-level processing all together. Most of them are due to adding some specific counters that require special processing. Figure 13b further breaks down all correlated changes related to device-level processing for Gen2 and Gen3. It shows that Gen3 has significantly reduced the correlated changes by 54.1%, 71.0%, and 100.0% (i.e., the second-to-last bar pairs) accordingly.

6.3 Benefits of change-driven Toposyncer

The first benefit is explicitly tracking changes in a centralized manner. Figure 14a shows the magnitude of the changes over



(a) No. of changes to Toposyncer. (b) Lagging reduction.

Figure 14: Change-driven Toposyncer.

time to Toposyncer. Note that this is much higher than the changes presented earlier, since it includes the changes of raw data for network states.

The second benefit lies in the efficiency and accuracy improvement to applications. We evaluate it using the lagging time, i.e. the time between the change happening and when changes are reflected in derived topology by Toposyncer. Figure 14b shows the topology derivation is much more timely: reducing 118.76s lagging time for ISIS interface updates, and 108.93s for BGP session state updates, averagely.

7 Lessons and Future Directions

We discuss our lessons from building PCAT and the opportunities for future research.

Efficiency vs. adaptivity. We work closely with vendors to improve the efficiency of data collection primitives at switches (similar to academic work on reducing memory usage and collection overhead with high accuracy [24, 26, 46]). However, pursuing efficiency brings us challenges on adaptivity. Different devices have different programming capabilities and resource constraints to adopt efficient algorithms. Introducing new primitives also adds diversity and dynamics to upper layers in the telemetry system. For example, we work with vendors to support a sophisticated micro-burst detection on hardware. However, if only a subset of switches supports this new feature, applications need complex logic to handle detected and missed micro-bursts. Thus we have a higher bar for adopting efficient algorithms due to adaptivity concerns.

To support diverse data collection algorithms, we need a full-stack solution with universal collection interfaces at switches and change-aware data processing and aggregation algorithms. Recent efforts on standardizing switch interfaces such as OpenConfig [3] is a great first step but does not put enough emphasis on standardizing telemetry interfaces. Recent trends on open-box switches (e.g., FBOSS [13]) bring new opportunities to develop adaptive telemetry primitives.

Trustful network telemetry. Telemetry becomes the foundation for many network management applications. Thus we need to know which data at which time period is trustful. However, building a trustful telemetry system is challenging in an evolving environment with many changes of devices, network configurations, and monitoring intents. Fast evolution also introduces more misconfigurations and software bugs. Explicitly tracking change cubes and exploring their dependencies in PCAT is only the first step.

We need more principled approaches for *telemetry verification and validation* across monitoring intents, data models, and collection jobs. Compared with configuration verification work [7, 35], telemetry verification requires quantifying the impact of changes to the measurement results. One opportunity is that we can leverage cross-validations across multiple counters covering the same or related network states or across aggregated statistics over time. For example, we collect power utilizations (watts) from both switches and power distribution units (PDUs). In this way, we can validate the correctness of these utilization counters by comparing the PDU value with the sum of switch values.

Telemetry systems are complex time-series databases. We can leverage provenance techniques [12] to support change tracking, data integration, and troubleshooting. One challenge is that we cannot build a full provenance system due to vendor-proprietary code and network domain-specific data aggregation algorithms. There are also unexpected correlation dependencies across data.

Integration between telemetry and management applications. Our production networks are moving towards self-driving network management with a full measure-control loop. PCAT shows that changes bring a new complexity to the measure-control loop. Control decisions not only affect the network state that telemetry system captures but also the telemetry system itself. For example, an interface change may affect a counter scope. A traffic engineering control change may affect data aggregation because traffic traverses through different switches. These telemetry data changes in turn affect control decisions. We need to identify solutions that can feed control-induced changes directly into the telemetry systems.

Another question is how to present large-scale multi-layer telemetry data to control applications. Rather than providing a unified data stream, control applications can benefit from deciding what time, at what granularity, frequency, and availability level for data collection and the resulting overhead and accuracy in the telemetry system. One lesson we learned is to have the telemetry data available when it is mostly needed. For example, the network’s aggregated egress traffic counter, which is collected at the edge PoPs, is a strong indicator of the business healthiness. To ensure its high availability, we need to give control applications the option to transfer the counter on more expensive out-of-band overlay networks. Moreover, we may extend the intent model to explicitly express the reliability-cost tradeoffs and adapt the tradeoffs during changes. We also need new algorithms and systems that can automatically integrate data at different granularities, frequencies, and device scopes to feed in control applications.

8 Related Work

Network evolvement. Several existing works have also pointed out the importance of considering changes. Both Robotron [41] and MALT [29] discuss it in the context of topology modeling, but miss the practical challenges of net-

work monitoring. [16] discusses network availability during changes, while we focus on telemetry systems during changes.

Other monitoring techniques. PCAT is a passive approach. Active measurement injects packets into the network [14, 17, 18, 34, 49], and they are complements to passive measurement. The design principle of PCAT to handle changes can be applied to existing monitoring systems [20, 25, 36, 44, 48, 50], languages and compilers [9, 19, 32, 33]. PCAT also benefits from recent software-defined measurement frameworks [25, 27, 32, 46, 48]. For example, similar to OpenSketch [48], PCAT frees network engineers from configuring different measurement tasks manually. PCAT’s intent model design borrows ideas from the query language in Marple [32]. There are many memory-efficient monitoring algorithms [22, 23, 26, 30, 46] that focus on the expressiveness and performance of network monitoring. They provide adaptivity but only to a limited type of new queries, resource changes, or network condition changes. Here, PCAT focuses on a broader set of adaptivity (e.g., adaptive to counter semantics changes, data format changes, and more).

Dependency in network management. Dependency graph has been widely used for root cause localization [5, 6, 37, 42, 43, 47, 50]. Statesman [40] captures domain-specific dependencies among network states. We share some similarities but use dependency to tackle the change propagation.

Techniques from database and software engineering. Data provenance [12, 15] encodes causal relations between data and tables in metadata. Several works [11, 45] apply provenance to network diagnosis. [8] proposes the change cube concept and applies it to real-world datasets. All the above works focus on data face-value. On the other hand, software engineering community studies the problem of how a change in one source code propagates to impact other code [21, 51]. Ours looks at changes from telemetry systems from both data, configurations, and code.

9 Conclusion

This paper presents the practical challenge of a monitoring system to support an evolving network in Facebook. We propose explicitly tracking changes with change cubes and exploring changes with a set of primitives. We present extensive measurements to illustrate its prevalence and complexity in production, then share experiences in building a change-aware telemetry system. We hope to inspire more research on adaptive algorithms and evolvable systems in telemetry.

Acknowledgments

We thank our shepherd Chuanxiong Guo and the anonymous reviewers for their insightful comments. Yang Zhou and Minlan Yu are supported in part by NSF grant CNS-1834263.

References

- [1] Express backbone. <https://engineering.fb.com/ata-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/>.
- [2] Introducing proxygen facebook c++ http framework. <https://code.fb.com/production-engineering/introducing-proxygen-facebook-s-c-http-framework>.
- [3] OpenConfig YANG model. <http://www.openconfig.net/projects/models/>.
- [4] Scribe. <https://github.com/facebookarchive/scribe>.
- [5] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 419–435, 2018.
- [6] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM Computer Communication Review*, 37(4):13–24, 2007.
- [7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.
- [8] Tobias Bleifuß, Leon Bornemann, Theodore Johnson, Dmitri V Kalashnikov, Felix Naumann, and Divesh Srivastava. Exploring change: a new dimension of data analytics. *Proceedings of the VLDB Endowment*, 12(2):85–98, 2018.
- [9] Kevin Borders, Jonathan Springer, and Matthew Burnside. Chimera: A declarative language for streaming network traffic analysis. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 365–379, 2012.
- [10] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. Simple network management protocol (snmp). Technical report, 1990.
- [11] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 115–128. ACM, 2016.
- [12] James Cheney, Laura Chiticariu, Wang-Chiew Tan, et al. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases*, 1(4):379–474, 2009.
- [13] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 342–356. ACM, 2018.
- [14] Cisco. Ip slas configuration guide, cisco ios release 12.4t. <http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipsla/configuration/12-4t/sla-12-4t-book.pdf>.
- [15] Mahmoud Elkhodr, Belal Alsinglawi, and Mohammad Alshehri. Data provenance in the internet of things. In *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 727–731. IEEE, 2018.
- [16] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 58–72. ACM, 2016.
- [17] Nicolas Guilbaud and Ross Cartlidge. Google localizing packet loss in a large complex network. Nanog57, Feb 2013.
- [18] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.
- [19] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 357–371, 2018.
- [20] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, volume 14, pages 71–85, 2014.
- [21] Ahmed E Hassan and Richard C Holt. Predicting change propagation in software systems. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 284–293. IEEE, 2004.

- [22] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 113–126, 2017.
- [23] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketch-learn: Relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 576–590, 2018.
- [24] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 404–421, 2020.
- [25] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 311–324, 2016.
- [26] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350, 2019.
- [27] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.
- [28] Chris Lonwick. The bsd syslog protocol. Technical report, 2001.
- [29] 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 {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 403–418, 2020.
- [30] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Scream: Sketch resource allocation for software-defined measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–13, 2015.
- [31] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 129–143, 2016.
- [32] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.
- [33] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. Compiling path queries. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 207–222, 2016.
- [34] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. detector: a topology-aware monitoring system for data center networks. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 55–68. USENIX Association, 2017.
- [35] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 953–967, 2020.
- [36] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network’s (data-center) network. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 123–137. ACM, 2015.
- [37] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive realtime datacenter fault detection and localization. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 595–612, 2017.
- [38] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 418–431. ACM, 2017.
- [39] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demaily, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter

- Xie. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, May 2015. USENIX Association.
- [40] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A network-state management service. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 563–574, 2014.
- [41] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 426–439. ACM, 2016.
- [42] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with path-dump. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 233–248, 2016.
- [43] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: active device and link failure localization in data center networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 599–614, 2019.
- [44] Mea Wang, Baochun Li, and Zongpeng Li. *sFlow: Towards resource-efficient and agile service federation in service overlay networks*. IEEE, 2004.
- [45] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 395–420, 2019.
- [46] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575. ACM, 2018.
- [47] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dshark: a general, easy to program and scalable framework for analyzing in-network packet traces. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 207–220, 2019.
- [48] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI*, volume 13, pages 29–42, 2013.
- [49] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252. ACM, 2012.
- [50] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.
- [51] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

APPENDIX

The first step of PCAT is to collect data from devices, which we call discovered data. There are three types of data including numeric counters, non-numeric states, and configurations. Table 4 shows the examples for each category.

Types	Categories & examples	Impact of software upgrades
Counters	<i>Device utilization:</i> CPU&memory utilization, routing table size, etc	Ambiguity between percentage and absolute values.
	<i>Device internal status:</i> Interface error counter, power supply temperature, fan speeds, linecard version, optical CRC error counter, etc	XML format gets changed; linecard version format changes from integer to string.
	<i>Packet processing counters:</i> Packet drops, errors, queue length, etc	Ambiguity of interface stats meaning.
	<i>Protocol counters:</i> BGP neighbor received routes, etc	General empty data error.
States	<i>Interface state:</i> Interface up, down, drained, configured IP address, MAC address, etc	Hex-decimal change causes MAC address retrieving error.
	<i>Protocol state:</i> BGP neighbor state, etc	State meaning ambiguity.
Configs	BGP policy, queuing algorithm, etc	Raw config format changed.

Table 4: Different discovered data in PCAT.

SwarmMap: Scaling Up Real-time Collaborative Visual SLAM at the Edge

Jingao Xu^{1*}, Hao Cao^{1*}, Zheng Yang^{1✉}, Longfei Shangguan²

Jialin Zhang¹, Xiaowu He¹, Yunhao Liu¹

¹School of Software, Tsinghua University ²University of Pittsburgh & Microsoft

Abstract

The Edge-based Multi-agent visual SLAM plays a key role in emerging mobile applications such as search-and-rescue, inventory automation, and industrial inspection. This algorithm relies on a central node to maintain the global map and schedule agents to execute their individual tasks. However, as the number of agents continues growing, the operational overhead of the visual SLAM system such as data redundancy, bandwidth consumption, and localization errors also scale, which challenges the system scalability.

In this paper, we present the design and implementation of SwarmMap, a framework design that scales up collaborative visual SLAM service in edge offloading settings. At the core of SwarmMap are three simple yet effective system modules — a change log-based server-client synchronization mechanism, a priority-aware task scheduler, and a lean representation of the global map that work hand-in-hand to address the data explosion caused by the growing number of agents. We make SwarmMap compatible with the robotic operating system (ROS) and open-source it¹. Existing visual SLAM applications could incorporate SwarmMap to enhance their performance and capacity in multi-agent scenarios. Comprehensive evaluations and a three-month case study at one of the world’s largest oil fields demonstrate that SwarmMap can serve 2× more agents (>20 agents) than the state of the arts with the same resource overhead, meanwhile maintaining an average trajectory error of 38cm, outperforming existing works by >55%.

1 Introduction

Visual simultaneous localization and mapping (SLAM) systems take video streams from one or multiple cameras as input, reconstructing the 3D map of environment while simultaneously determining the position and orientation of cameras with respect to their surroundings [29, 34, 36]. With the size of the mapping area expanding rapidly, collaborative visual

SLAM that involves multiple agents has been attracting growing interest from both academia and industry [25, 39, 40, 44]. For instance, Amazon, JD, and Alibaba have deployed dozens of picking and sorting robots in their logistics warehouses to save labor cost [45]; DJI and Amazon have also been developing drone grouping and swarming technology for urban modeling, express delivery, and industrial inspection [12]. In these scenarios, each agent has to conduct not only the localization but mapping tasks in real-time due to (i) upper layer applications require the latest updated environment map to perform the subsequent maintenance and scheduling tasks, especially in those dynamic environments; and (ii) since the two modules are tightly coupled, an agent also relies on a high-quality on-board map for a better localization performance and vice-versa [3, 47].

The SLAM agents profile the environment with their cameras, exchange data with each other, and execute vision tasks in real-time, with a significant computation overhead. The limited computation resource on the agent soon becomes the bottleneck, impairing system accuracy [3, 40, 47]. *Edge-offload* has emerged as a promising alternative due to the following two reasons. First, by offloading bulky tasks to edge devices, the agents only need to run light-weight and time-sensitive jobs locally, which effectively mitigates on-board resource shortage [3, 47]. Second, by fusing and further optimizing the visual map globally at a centralized edge device, map information that is originally unavailable to each other can be easily shared among agents [39, 40]. This will benefit collaborative missions such as collision avoidance and path planning.

Albeit inspiring, the growing number of agents brings new issues that challenge the scalability of edge-based real-time collaborative visual SLAM systems (§2.2):

- **Map synchronization stresses the network bandwidth.** Mobile agents like drones and robots heavily rely on wireless links to communicate with an edge device. However, wireless spectrum is a limited and overcrowded resource. Streaming large volumes of map data over wireless links will soon saturate the medium and cause significant delays.

- **FCFS-based job scheduling impairs the localization ac-**

^{*}Zheng Yang (hmilyyz@gmail.com) is the corresponding author. Jingao Xu and Hao Cao are co-primary authors.

¹Code and data at <https://github.com/MobiSense/SwarmMap>.



Figure 1: Industrial inspection is carried out by 10 drones and 2 autonomous vehicles in one of the world’s largest oil-field ($>170\text{km}^2$) in the Middle East. These agents are coordinated by SwarmMap that runs on an Nvidia AGX Xavier edge server.

curacy. An edge device has to processes large volumes of requests from agents, which may cause significant delays to latecomers (i.e., those requests positioned in the tail of the queue). However, agents in different states are not equally sensitive to the queuing delay. The conventional first-come, first-served (FCFS) pipeline will exacerbate the localization error on those time-sensitive agents.

- **Map expansion exacerbates the memory footprint.** The size of the global visual map increases sharply with a growing number of agents, which is likely to exceed the limited memory capacity allocated to SLAM tasks by an edge node, causing memory overflow.

However, the current practice of edge-offload focuses primarily on computation-oriented task partitioning [3, 8, 23, 40, 47]. They fail to address the data explosion and its impact on transmission, scheduling, and storage. Hence these pioneer designs cannot scale with the sheer size of the real-time collaborative visual SLAM systems.

In this work, we present SwarmMap, a framework to scale up the real-time collaborative visual SLAM services at resource-constrained edge devices. SwarmMap does not innovate visual SLAM algorithms. Instead, it proposes functionality and resource abstractions of existing SLAM algorithms and provides additional system services to enhance system scalability. Hence, most variations of collaborative visual SLAM systems can take advantage of our design. With SwarmMap, the upper-layer user can outsource agent task scheduling and processing instead of understanding every detail of SLAM algorithms to manually adapt. SwarmMap contains three key plug-in modules, as described below.

First, we design a Map Information Tracker (*Mapit*) to maintain map data consistency between the agents and the edge while remarkably saving network bandwidth. Unlike existing methods that transfer bulky map data with each other [39, 40], *Mapit* records the operations associated with the map modification on the agent and transmits these operations to the edge. The edge node then follows these operations to update its local map. This allows the map synchronization

between them at the minimum bandwidth consumption even compared with state-of-the-arts (e.g., CarMap [2]).

Second, we introduce a SLAM-specific task-aware scheduler (*STS*) that prioritizes requests based on the status of their producer (i.e., agent). The *STS* scheduler runs on both the agent and the edge. The agent *STS* evaluates agent status around the clock and updates this information with the edge through heartbeat packets. The edge *STS* designs a multi-level queue to ensure those urgent tasks will be processed timely.

Third, we propose a Map Backbone Profiling (*MBP*) technique to alleviate the storage overhead while retaining the mapping accuracy. This technique is based on an observation that the data quality among different agents’ maps can be balanced by elements in co-visible areas. We propose a set of metrics to detect high-quality map elements and use them to offset those low-quality counterparts, thereby elevating the overall map quality. Applying model compression to this high-quality map allows us to remove large portions of redundant map data without sacrificing the map accuracy.

We evaluate SwarmMap on a testbed consisting of 4 Nvidia Jetson boards, 4 smartphones, 4 DJI RoboMasters, and 4 drones. Following the standard SLAM evaluation pipeline [2, 6, 28, 47], we further compare SwarmMap with two state-of-the-art (SOTA) edge-assisted multi-agent SLAM systems (CCM-SLAM [40] and Multi-UAV [39]) on three gold-standard SLAM datasets (TUM [11], KITTI [10], and EuRoC [9]) as well as a self-labeled dataset collected at a 22,927 sqft shopping mall. We also compare SwarmMap with CarMap [2] and Sum-Map [27] to evaluate each functional module in SwarmMap. Our head-to-head comparison shows that SwarmMap can serve 2 \times more agents than these SOTA systems with the same resource overhead, meanwhile maintaining an absolute trajectory error within 38cm when serving 20 agents, outperforming these SOTA systems by >55%.

Real-world deployment. We have developed a real-time collaborative visual SLAM system based on SwarmMap and deployed it in one of the world’s largest oil-field ($>170\text{km}^2$) for industrial inspection (shown in Fig. 1). Our system con-

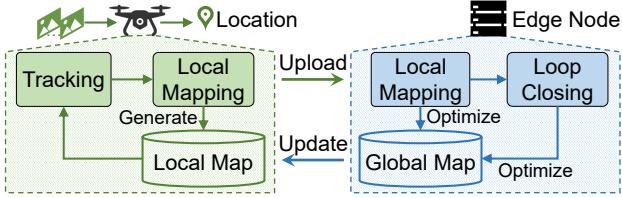


Figure 2: Workflow of existing edge-assisted SLAM [3, 47]

sists of twelve agents that communicate with an Nvidia Jetson AGX Xavier [46] edge node through Wi-Fi mesh networks. A three-month pilot study shows that SwarmMap achieves an average localization accuracy of $0.36m$. The link throughput and RAM consumption are below 17MB/s and 26GB respectively, meeting inspection demands within the constraints of available resources.

In summary, this paper makes three contributions. First, we quantify the scalability challenges of deploying real-time collaborative visual SLAM at the edge to motivate framework support. Second, we design and implement SwarmMap as a framework to address the scalability issues spanning from communication, computation, to storage. As far as we are aware of, SwarmMap is the first system solution to scale up the collaborative visual SLAM in edge settings. Third, we deploy SwarmMap in one of the world’s largest oil fields for industrial inspections in the Middle East. Our three-month pilot study demonstrates that SwarmMap makes a great process towards fortifying multi-agent collaborative visual SLAM to a fully practical system for wide deployment.

Contribution to the community. We implement SwarmMap as a software package of the robot operating system (ROS [26]), the dominating OS in the robotics field. We believe SwarmMap can provide a collection of tools for both academia and industry, and further enable fast prototyping of visual SLAM-based applications in multi-agent scenarios.

2 Background and Motivation

The data volume scales with the number of agents, and the need for framework support arises from the excessive bandwidth consumption and memory footprint caused by the data explosion. We discuss these in detail in this section.

2.1 Edge-assisted visual SLAM systems

The visual SLAM consists of multiple sub-tasks with diverse workloads. Edge-offload places those bulky tasks to an edge server, leaving an agent light-weight and time-sensitive jobs. The agent can thus run visual SLAM in real-time. We use ORB-SLAM2 [29], a top-ranked open-source visual SLAM system, to illustrate the SLAM operations under edge settings (refer to Fig. 2).

Front-end. Mobile agents run *Tracking* and part of the *Local Mapping* module locally. The *Tracking* module extracts 2D ORB feature points from each video frame and instantly estimates the pose of onboard camera(s) based on the geometry relationship between these feature points and the

pre-constructed local map (i.e., a set of 3D map-points and keyframes² in which they appear). As the mobile agent moves, the *Local Mapping* module updates the local map timely.

Back-end. Due to high computation costs, the optimization part of the *Local Mapping* module is offloaded to the edge device, where the bundle adjustment (BA) algorithms [42] kick in to improve the pose and 3D location accuracy of those newly generated keyframes and map-points. The edge server also runs a *Loop Closing* module to detect repeated paths and leverage them to re-calibrate the global map.

Data transfer in-between. To improve the map accuracy, each agent periodically sends keyframes and map-points to the edge server for fine-grained optimization. The optimized visual map is then streamed to the clients.

2.2 The scalability issues

As more agents get involved, running real-time collaborative visual SLAM on edge environment becomes increasingly complex, facing several challenges: (i) the frequent data transfer between agents and edge is likely to saturate wireless links, causing significant delays; (ii) the queueing delay on edge node exacerbates localization errors; (iii) the data volume grows sharply, threatening the data storage at the edge node. We discuss these issues below.

C1: Excessive bandwidth consumption. The life-cycle of a collaborative visual SLAM system consists of cold-start and maintenance two sessions. In the cold-start session, the agents transfer all observed keyframes and map-points data to the edge server. The edge server then generates a global map of the entire space and optimizes the local map for each agent. Once the global map generation has been completed, the SLAM system enters the long-term maintenance session during which each agent regularly revisits each site and calibrates the mapping offset. However, since map elements are tightly coupled, a minor modification on a single map element will spread to many other elements. This will cause a significant amount of data transfer in the maintenance.

To reduce bandwidth consumption, recent works [2, 40] design compact map representations and transfer the difference before and after map element calibration (as opposed to transferring the entire calibrated map element [39, 47]). Although these systems can effectively reduce bandwidth consumption in the cold-start session, they encounter two issues in the maintenance session due to the frequent map updates: (i) *extra computation overhead*. The acquisition of element-level differences requires pair-wise map feature comparison across the entire map. This will lead to extra computation workload pressure on resource-limited mobile agents; and (ii) *limited data volume reduction*. Since a minor change on an element will spread to a batch of coupled elements, the volume of data to be transferred is still bulky.

²Keyframes are a subset of selected frames. Each keyframe stores the camera pose, the map-points it observed, and the co-visibility relationships with other keyframes.

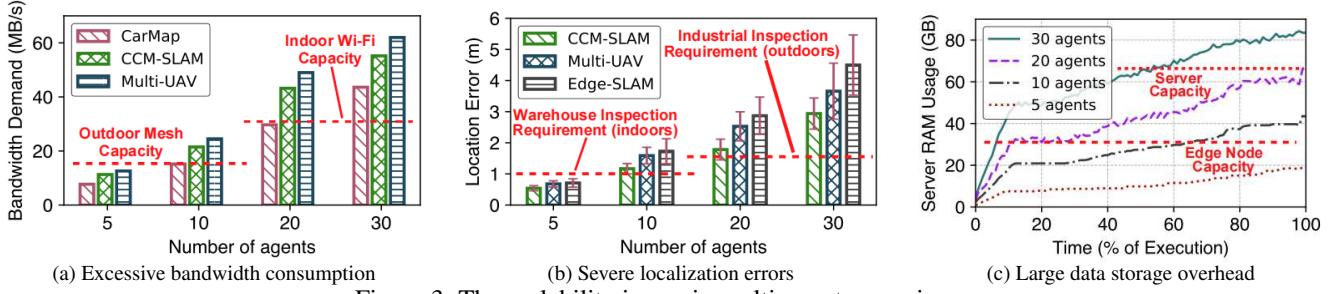


Figure 3: The scalability issues in multi-agent scenarios.

To validate our analysis, we measure the bandwidth requirement of three state-of-the-art (SOTA) systems in different number of agents settings. The results are shown in Fig. 3a. Compared with the vanilla Multi-UAV [39], we observe that CarMap [2] and CCM-SLAM [40] can effectively reduce the transmission workload during map synchronization. However, when serving more than ten agents, both systems still produce excessive wireless traffic that can easily go beyond the link capacity³; thus, significant system delays are expected.

C2: Severe localization errors. Under the edge settings, the localization accuracy of an agent highly depends on the quality of the local map which is optimized at the edge side. Typically, an agent needs to periodically (within 5s) send optimization requests to the edge server for every 3-5 newly generated keyframes [3, 47]. As the number of agents scales, the concurrent requests from different agents block at the edge node’s processing pipeline, resulting in excessive queuing delays. Consequently, some agents get their optimization tasks done untimely, causing severe localization errors. This situation is worsened by the fact that agents in different running states (e.g., flying speeds, self-tracking qualities) are not equally sensitive to the waiting delay. Recent multi-agent collaborative SLAM solutions focus on map fusion and optimization on edge or cloud servers, but ignore the task queuing issue for each agent. The conventional first-come, first-served (FCFS) scheduling will inevitably exacerbate the localization error on those task-sensitive agents (demonstrated in §5.3).

We measure the localization error (in m) of three related works in a different number of agents settings. The results are shown in Fig. 3b. Considering the accuracy requirement from a broad range of SLAM applications, we treat 1m and 1.5m as acceptable localization errors for indoor (warehouse inspection) and outdoor (anomaly detection) scenarios. Evidently, all these three systems fail to meet the localization requirement when serving more than 5 and 10 agents indoors and outdoors respectively, leaving room for improvements.

C3: Large data storage overhead. The global map maintained by the edge server contains large redundancy due to the following two reasons. First, to ensure the inspection efficacy, different agents will re-visit the same area at certain intervals, causing significant path duplication. Second, to

³The measurement shows the maximum throughput in an outdoor mesh and an indoor 2.4 GHz Wi-Fi network is 15MB/s and 30MB/s, respectively.

complete the 3D map reconstruction, different agents have to share a co-visible area, resulting in bulky data redundancy. As the number of agents grows, the data redundancy increases sharply, and the data volume is likely to exceed the limited memory capacity of the edge node.

We set up an edge-based collaborative visual SLAM testbed using a commercial edge device Nvidia Jetson AGX Xavier (with 32GB RAM and costs \$599) and measure its RAM usage in different numbers of agent settings. We repeat the measurement on a powerful server with 4× higher storage capacity (i.e., Dell PowerEdge T630 with 128GB RAM and costs \$6,899) for comparison. The results are shown in Fig. 3c. In accordance with our analysis, as the system proceeds, the RAM usage increases rapidly and soon saturates the memory capacity of both the edge node and the high-end server. This limitation is worsened by the mismatch between the limited storage capacity of the edge node and the growing fidelity of video streams (i.e., 4K or 8K videos). Such high memory demand limits the maximum number of agents to five, which sets a strong barrier for the practical deployment of the edge-based collaborative visual SLAM system.

Due to the device heterogeneity (e.g., cameras on drones and robots may differ drastically in video resolution and frame rate) and diverse running status, the quality of maps provided by different agents may vary largely. An ideal map compression should remove those low-quality redundancy while retaining the high-quality counterpart. However, existing works ignore such difference when compressing the map data [27, 32, 43], resulting in degraded SLAM performance (details in §5.3).

2.3 SwarmMap: System goals

SwarmMap takes a solid step forward in solving these scalability issues. We list the system goals below.

Goal 1: Functionality and resource abstraction. SwarmMap should provide functionality and resource abstractions of existing SLAM algorithms. This allows any variation of map-point- and keyframe-based collaborative SLAM algorithms to take advantage of SwarmMap.

Goal 2: Plug and play. SwarmMap should be implemented as a plug-in module, exposing well-defined APIs to end-users for adaption. This avoids the deeply embedded manual code changes that may again challenge the system’s scalability.

Goal 3: Resource overhead reduction. SwarmMap should effectively reduce the resource overhead spanning data storage, client-edge communication, and task scheduling while ensuring the precision and real-time performance.

3 Design

In this section, we first describe the high-level system architecture and then present each module design in SwarmMap.

3.1 System overview

SwarmMap is a framework design to scale up collaborative visual SLAM service in edge offloading settings. To achieve this goal, we make the following layer-wise functionality and resource abstractions: (i) *agent layer*, where each agent localizes itself and builds surrounding local maps in real-time; (ii) *network layer*, which enables communications and data interactions between mobile and edge for map synchronization; and (iii) *edge layer*, which fuses, optimizes, and maintains the global map. This layer-wise abstraction provides a clear view of map data transfer, processing, and storage in SLAMs.

Key functional modules. SwarmMap designs three plug-in modules to address the resource overhead and scheduling issues across these three layers.

- The *Mapit* (Map Information Tracker) module tracks system operations associated with map data calibration. It then transfers these operations to the peer(s) for map synchronization (§3.2).
- The *STS* (SLAM-specific Task Scheduling) module optimizes the batch request execution and manages the resource allocations among multiple agents (§3.3).
- The *MBP* (Map Backbone Profiling) module compresses the map data uploaded by individual agents while ensuring the overall mapping accuracy (§3.4).

SwarmMap Architecture. Fig. 4 shows the system architecture. SwarmMap shares similar edge-based architecture with previous works and provides extra system support on both the mobile agent and edge server side, as discussed below.

- On the mobile agent side, SwarmMap tracks the run-time status of each agent through a light-weight evaluation-based mechanism *STS* (mobile part). It then follows a dedicated information exchanging protocol *Mapit* to communicate and update map elements with the edge server.
- On the edge side, the edge node prioritizes the agents’ requests by *STS* (edge part) based on their run-time status. It then takes into account the data quality of maps reported by individual agents and extracts a lean presentation of the overall map through a map backbone profiling algorithm (*MBP*). Finally, the optimized and compressed map backbones will be sent to each mobile agent by *Mapit*.

3.2 Mapit: Map Information Tracker

The inevitable frequent map data synchronization between clients and edge consumes large bandwidth in both cold-start

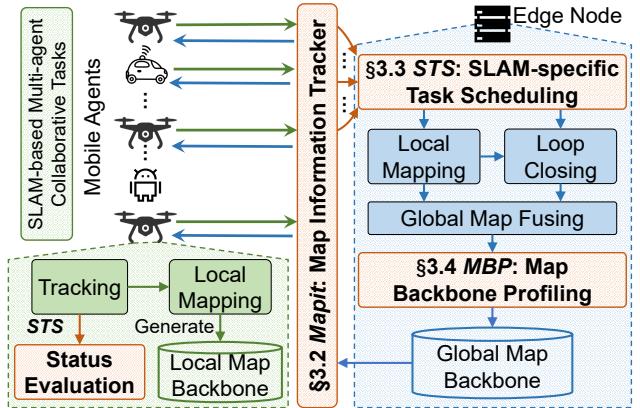


Figure 4: System architecture of SwarmMap. Compared with the conventional edge-based visual SLAM architecture, the added plug-in modules are highlighted in orange.

and maintenance sessions, circumscribing the system capacity (i.e., the number of supported agents). Recent works (e.g., CarMap [2] and CCM-SLAM [40]) propose a compact map representation that greatly reduces the data transfer in the cold-start session. However, their effectiveness fails to translate to a sufficient reduction in the maintenance session (§2.2-C1). Therefore, in SwarmMap we focus on the data transfer reduction in the maintenance session.

Our design is based on an observation that the map change on one side can be reproduced on the other side (e.g., agent vs. edge) by solely transferring the map change operations. This enables a light-weight map synchronization by avoiding transferring massive map-point data and the bulky geographical descriptors such as their spatial locations, features, observation relationships with keyframes [28]. Compared with the current practice, our design also achieves higher synchronization efficiency because it does not require a pair-wise map element comparison, which leads to extra computation workload pressure on resource-limited mobile agents.

To realize this basic idea, we design *Mapit*, a light-weight map information tracker to automate the operation tracking and reproducing on mobile and edge. *Mapit* runs as a daemon on both sides, monitoring the SLAM function calls and logging corresponding map operations (e.g., move a map-point by 2cm). It then transfers this log to the agent (or the server), based on which the agent reproduces these operations locally. The map data are synchronized at the end.

The *Mapit* package periodically⁴ synchronizes the map operation logs, and consists of five atomic operations: *add*, *aggregate*, *push*, *merge*, and *pull* (shown in Fig. 5).

① **Mapit add.** The atomic operation *add* registers a hook for each SLAM function call (listed in Table 3) and maintains a recording queue. Whenever an important function is called, an operation record containing its name, parameters, and influence on map elements is *added* to the operation queue.

⁴Similar to current practice [2, 3, 39], we empirically set the period to 2s.

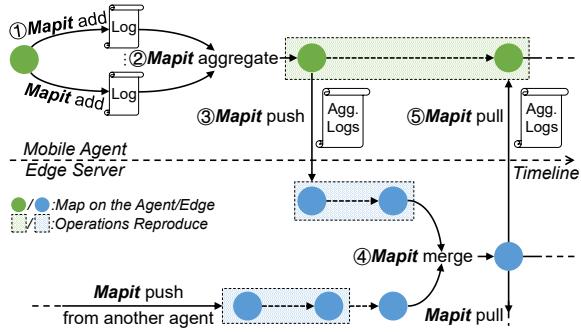


Figure 5: Workflow of the *Mapit*.

② Mapit aggregate. At the end of each period, *Mapit* aggregates the records in the operation queue to reduce their size. The intuition is that some removals or merges on certain types of functions will generate equivalent effects. For instance, if a function changes the location of a map-point and is marked as overwritten, we only need to focus on the latest record of it and ignore all the previous operations on the map-point. As for those marked as stackable, the implication is that records about modifying a same element can be merged by parameters. In this way, *Mapit* produces a minimal set containing necessary information.

③ Mapit push. After aggregating records, the atomic operation *push* on an agent sends the packed records to the edge server. By reproducing these operations, the map maintained on edge keeps synchronized with the ones on the client.

④ Mapit merge. On the edge server, the *merge* module periodically checks if there exists an overlap between the maps uploaded by individual agents and the global map. Once an overlap is detected, different maps will be coordinated and fused by the upper-layer SLAM algorithms (e.g., Sim3 optimization algorithm [28]). The map fusion process will operate and update some map elements, and hence the *merge* module also records these operations on the map elements in the same way as *add* and *aggregate*.

⑤ Mapit pull. The *pull* module can be treated as the reverse operation of *push*. It requests aggregated map modification logs generated by map optimization and *merge*, from the edge server to the agent. Additionally, if the global map has already been created (i.e., the whole system is in the maintenance session), *Mapit* will also transfer a set of closest map-points (e.g., associated with the next 5 keyframes) to the agent in the *pull* process. The benefit of this strategy is to enhance the agent's localization performance since these map-points with a high probability of appearing in the future would provide prior information for the *tracking* module on the agent side.

3.3 STS: SLAM-Specific Task Scheduling

As more agents get involved in SLAM systems, processing agents' requests (e.g., local map optimization) can cause excessive queuing delays. Since agents in different running states are not equally sensitive to the waiting delay, conventional FCFS scheduling may exacerbate localization errors

on time-sensitive agents and hurt SLAM performance (§2.2-C2). To our best knowledge, there is still a lack of scheduling strategy tailor to multi-agent SLAM tasks.

To address this issue, we introduce *STS* – the first SLAM-Specific Task Scheduler that guides the edge to strategically prioritize requests. Specifically, *STS* divides agents into emergency and non-emergency groups based on the agents' status. It timely reorders the requests based on the following principles:

- (i) Prioritizing requests from agents in the emergency group.
- (ii) Among those non-emergency agents, *STS* prioritizes requests from agents that can provide higher information gain for global map construction or optimization.

The first principle aims to prevent each agent from losing self-tracking, and the second is for achieving a better overall global mapping performance. We propose a set of metrics to characterize the agent status and design a multi-level queue to schedule the requests from agents.

3.3.1 Agent Status Evaluation and Updating

Agent side. Each agent regularly updates its status with the edge by sending heartbeat packets. Since both environment and device dynamics may fluctuate violently during an agent's movement, the heartbeat interval should be shorter than the agent's request interval (i.e., 2s). In SwarmMap we expose the heartbeat setting (100ms by default) to end-users so that they can easily adapt to different environment settings. We define three variables that can fairly reflect an agent's status:

- **Tracking state:** a 1-bit Boolean value shows whether an agent is traceable or not. An agent's tracking state is set to LOST if its latest ORB feature maps cannot well match the local feature map. This variable is provided by the *tracking* module in many visual SLAM systems [29].
- **Velocity burst:** a 1-bit Boolean value shows whether an agent's speed changes abruptly or not. An abrupt change of velocity may result in motion blur in videos and make it hard for clients to extract visual features. In SwarmMap, we set the variable *Velocity burst* to True if the current moving speed is 20% greater than the averaged speed over the latest N frames, where N is a variable exposed to end-users. $N = 10$ by default.
- **Tracked map-points number:** an 8-bit variable represents the number of map-points observed by an agent. A larger number indicates the *tracking* module is running more stable.

Server side. Due to the heterogeneous device capability (e.g., cameras on different agents may differ in resolutions) and diversified trajectory, each agent contributes unequally to global map construction and optimization. SwarmMap prioritizes requests from those agents that can provide higher information gain for global map construction and optimization. To this end, we design the following two metrics to measure the information gain of each agent:

- **Map-point score (MS)** is defined as the average score of all map-points observed by an agent (the way to calculate the map-point score will be introduced in §3.4). A higher average

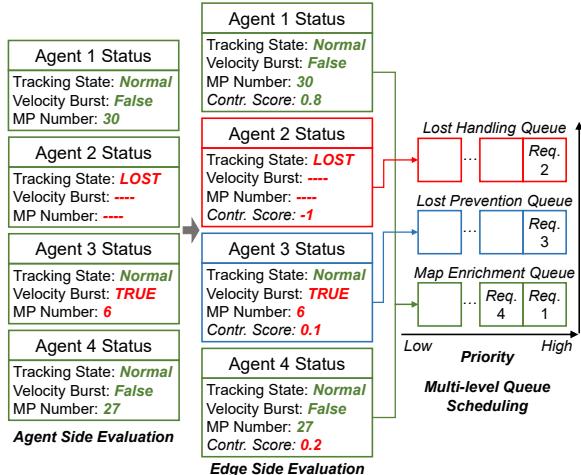


Figure 6: Workflow of the STS with an example.

score reflects that the current position is likely to have been visited before. On the contrary, a lower score indicates the agent is exploiting new or partially observed areas. Hence, *STS* prioritizes tasks with a lower map-point score.

- **Map elements generation speed** (MG) characterizes the number of unobserved map-points and keyframes uploaded by the latest *mapit push* operation. An agent with a higher map element generation speed contributes more to the edge’s global map generation and optimization.

STS normalizes each metric and computes each agent’s *contribution score* as normalized MG - normalized MS.

3.3.2 Multi-level Queue Scheduling

On the edge side, *STS* designs three queues with different priorities to facilitate agent request scheduling.

- **Lost Handling Queue**. If an agent’s tracking state is marked as LOST, *STS* will push its request into this queue.
- **Lost Prevention Queue**. If an agent has a velocity burst and merely tracks few map-points, it may become prone to LOST, and *STS* will push its request into this queue.
- **Map Enrichment Queue**. For those agents with stable running status (i.e., without the risk of losing self-tracking), *STS* will push their requests into this queue and sort them by their mapping contribution scores.

The lost handling queue owns the highest priority, followed by lost prevention queue and map enrichment queue. Upon the reception of an agent’s request, *STS* inserts this request into one of these three queues based on the agent’s tracking status and mapping contribution. The back-end SLAM algorithm pops requests from queues based on their priority.

We take Fig. 6 as an example to explain the job scheduling in SwarmMap. Suppose there are four agents in the system, with agent 2 in lost tracking status and agent 3 facing the velocity burst issue. *STS* will push agent 2 and 3’s requests into the lost handling and prevention queue, respectively. The request from agent 1 and 4, two agents not in emergency states, will be pushed into the map enrichment queue. Since

agent 1’s mapping contribution score is higher than agent 4, the request from agent 1 will be put at the head of the queue. The edge processes these requests in the order of 2-3-1-4.

3.4 MBP: Map Backbone Profiling

The global map maintained by the edge node contains large redundancy (§2.2-C3). Due to the device heterogeneity (e.g., the onboard cameras may differ in resolution and frame rate) and diverse running status, the quality of maps contributed by different agents may vary largely. Existing map compression works [6, 13, 14] ignore such difference, resulting in information loss and hence degraded performance. The relevant works, CarMap and CCM-SLAM, design lean map representations to reduce the transmitted data volume for a faster map synchronization. However, they still need to reconstruct the huge global map through these compact representations on both mobile agent and edge node. Therefore, the memory footprint remains high when more agents are connected.

To address this issue, we introduce a map backbone profiling (*MBP*) algorithm. Unlike the current practice, we do not greedily remove redundant map elements in co-visible areas. Instead, we first leverage these redundant elements to generate a series of virtual keyframes and use them to improve those low-quality map segments. Once the overall quality of the global map got improved, we can thus compress the global map without compromising the mapping quality.

MBP first evaluates the quality and importance of each map element. It then (i): finds high-quality map-points that could be leveraged to generate virtual keyframes; (ii): searches for low-quality map segments that need to be improved; and (iii): improves the overall map quality by inserting virtual keyframes to those low-quality map segments. Finally, *MBP* operates map compression on the balanced global map.

3.4.1 Map Element Evaluation

Map-point evaluation has been extensively studied in related works [14]. The gold-standard metrics include the observing path length, maximum observing distance, maximum observing angle, and mean re-projection error. We borrow these metrics (detailed in §A.2) to evaluate a map-point and propose three new metrics to adapt to collaborative scenarios:

- **Observed number** represents the number of keyframes, in which the map-point is observed, across the entire global map. A higher score indicates multiple agents can observe a map point over a long period.
- **Update frequency** is defined as the total number of times the map-point was modified or updated by all agents in the last round of *Mapit push* operations. Map-points with high update frequency suggest a potential hot spot in a trajectory.
- **Moving velocity** records the speed of a mobile device when it generates the map element. A higher score indicates a potential blurriness that may influence the stability of the map-point. We take its negative value to evaluate the map-point score.

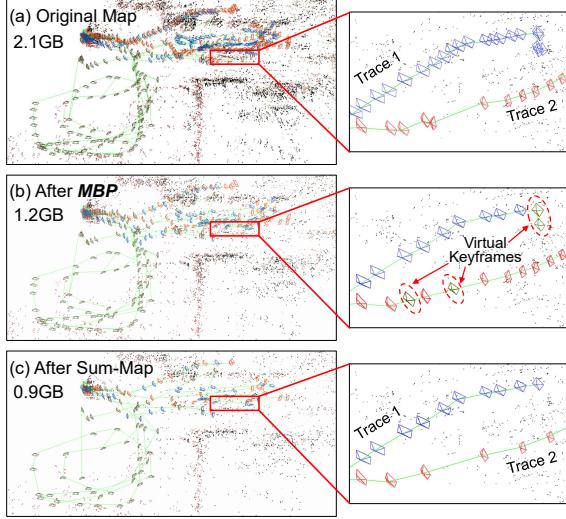


Figure 7: A running demo of *MBP*. The left column shows the map elements uploaded by different agents, while the right column presents partial zooming-in maps. The dotted keyframes in (b) are the synthetic virtual keyframes.

MBP normalizes each metric by its maximum value. We then define the score of a map-point as the sum of all normalized metrics values. The score of a keyframe is the sum of all observing map-point scores.

3.4.2 Map Backbone Generation

The map backbone generation consists of two steps: virtual keyframe generation and map compression.

Virtual keyframe generation. The trajectory of an individual agent is first segmented with the awareness of where overlaps occur. The quality of each map segment is defined as the sum of all map element scores (i.e., scores of all keyframes and map-points) within it. For each map segment with low quality (e.g., its score is in the bottom 20%), *MBP* search for high-quality map-points in its neighborhood (i.e., within 60° field-of-view of its keyframes) even though the original keyframes do not observe these map-points. Furthermore, *MBP* synthesizes virtual keyframes that could observe these high-quality map-points, and the pose (i.e., spatial location and orientation) of each keyframe can be calculated by the ICP algorithm [38] and optimized by BA [42]. Since the virtual keyframes only consider whether a map point is good enough regardless of which agent uploads it, they can supplement those low-quality segments.

Map compression. Once the quality of map segments is more balanced, *MBP* performs the similar map compression algorithm proposed by Sum-Map [27], eliminating redundancy by generating an enhanced minimum spanning tree across the global map. In addition, we introduce an extra optimization goal that guides the spanning tree to cover as many high-quality map elements as possible.

Fig. 7 compares the map compression performance of *MBP* and Sum-Map. Map elements from different agents

are marked in red, blue, and brown in the figure. Although Sum-Map obviously reduces the map size, it neglects the map quality difference, making the compressed map of trace 2 too sparse and harming the SLAM performance. In contrast, with reducing the map size by nearly half, *MBP* inserts several virtual keyframes, balancing the map quality among different agents and ensuring mapping accuracy.

4 Implementation

We implement SwarmMap as an open-source package and make it compatible with ROS [26]. It contains 18,000 LOC (line of C++ code). SwarmMap is built upon ORB-SLAM2 [29], the top-ranked open-source SLAM algorithm that has been widely used by both research and industry communities. Our implementation avoids modifications on SLAM functions (e.g., tracking, local mapping, loop closing). This allows any variation of ORB-SLAM algorithms such as DynaSLAM [5], ORB-SLAM3 [7], as well as other map-point-and keyframe-based collaborative SLAM algorithms (e.g., Multi-UAV [39], C-ORB [22], CCM-SLAM [40]) to take advantage of SwarmMap (demonstrated in §A.5). Additionally, we also expose well-packaged APIs to facilitate users to modify some parameters (map synchronization period, status evaluation metrics, etc.) in SwarmMap according to specific upper-layer applications. A high-level abstraction of SwarmMap’s implementation is detailed in §A.3.

5 Evaluation

In this section, we first present the experimental methodology (§5.1), followed by the overall performance of SwarmMap compared against SOTA systems (§5.2). We then conduct an ablation study to understand each functional module in SwarmMap (§5.3). Further, we demonstrate the portability of SwarmMap by plugging it into baseline SLAM systems (§A.5).

5.1 Experimental Methodology

Field studies. We deploy 12 agents including 4 smartphones, 4 drones, and 4 mobile robots on a 22,927 sqft shopping mall. These agents collaboratively localize themselves and mapping the environment in real-time. The ground truth is obtained through the Kinect 360 RGB-D and Opti-Track [33] cameras. We also build a dataset using these video streams for trace-driven evaluation.

Trace-driven evaluations. Following the conventional visual SLAM evaluation methodology [2, 22, 40, 47], we also conduct comprehensive trace-driven evaluations based on public SLAM datasets (KITTI [10], EuRoC [9], and TUM [11]) and the handcrafted dataset mentioned above. The characterization of three public datasets is summarized in Table 4. In our evaluations, the movement speed of mobile agents various significantly, ranging from $0.5m/s$ (indoor DJI RoboMasters) to $15m/s$ (ourdoor vehicles), representing the status of devices in

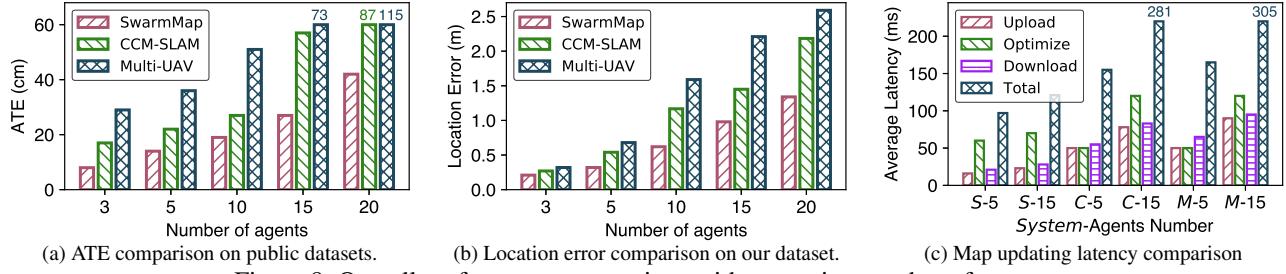


Figure 8: Overall performance comparison with a growing number of agents.

real world usage. Similar to the standard collaboration SLAM evaluation pipeline [19, 39, 40], we cut the video stream into overlapped segments and feed them to different agents to emulate the multi-agent scenario.

Edge Setup. Most of the previous works cannot be deployed on a resource-constrained edge node to support large numbers of agents because they consume a considerable amount of network bandwidth and edge computational resources (§2). We thus use a powerful server, which is equipped with an Intel(R) Xeon(R) CPU E5-2620v4 of 2.10GHz main frequency and 64GB RAM running Ubuntu 18.04, to explore the capacity of these systems and compare them with SwarmMap. The agents communicate with the server through 2.4 GHz and 5 GHz Wi-Fi links in the shopping mall and our laboratory. The maximally achievable link throughput measured with iperf3 is 27.4MB/s and 46.1MB/s, respectively.

Metrics. We use *absolute trajectory error* (ATE, in cm) to evaluate SLAM accuracy on the three public datasets while adopting *location error* (in m) to evaluate the positioning accuracy in field studies and our handcrafted shopping mall dataset. ATE is a golden metric for evaluating the tracking performance of SLAM algorithms [11]. Since ATE pre-calibrates the generated trace with the ground-truth trajectories before measuring the absolute errors, it achieves fewer errors than the actual location errors. To evaluate system overhead, we count the *bandwidth demand* (in MB/s) of all participants in the system (defined as the sum of the average volume of data transferred per second by all agents). Similar to previous works [40, 47], we store the global map in RAM rather than SSD during system operation for faster map recall and update. We hence record the *RAM usage* (in GB) on the edge server to measure the memory consumption.

Map updating latency. Similar to previous works such as Edge-SLAM [3] and CCM-SLAM [40], SwarmMap adopts the same edge-assisted architecture where the *tracking* task is running locally on the agents. This allows an agent to localize itself in real-time (i.e., >30 fps with camera rate). We thus take the *map updating latency* (in ms)—the delay until the agent gets the latest optimized map from the server—as the metric to evaluate the real-time performance of map updating in SwarmMap. Map updating latency takes into account both the map synchronization and optimization latency.

5.2 Overall Performance Comparison

We first compare SwarmMap with CCM-SLAM [40] and Multi-UAV [39], two most relevant SOTA edge-based multi-agent SLAM systems, to evaluate the overall performance.

5.2.1 Accuracy Comparison

We first evaluate the average ATE and location error in a different number of agent settings. The results are depicted in Fig. 8a and Fig. 8b. As seen, SwarmMap achieves the best tracking and localization performance in all scenarios. Compared with related works, SwarmMap reduces ATE by > 30%, 20%, 20%, 50%, 55% for scales with 3, 5, 10, 15, 20 agents, respectively. The location errors are also significantly degraded by >40% when serving more than 10 agents. On the other hand, the performance of CCM-SLAM and Multi-UAV degrades remarkably with the growing number of agents. (i.e., the ATE and location errors expand 3× and 7× respectively from 3 to 20 agents). When serving more than 10 agents in the shopping mall, they fail to guarantee that the average location error of each client is within 1.5m, which is typically the localization precision requirement for indoor drones [48]. In contrast, SwarmMap can still bound ATE and location error within 40cm and 1.4m even serving 20 agents. Generally speaking, above delightful results come from the fact that the localization performance of each agent highly depends on the quality of the on-board maintained local map [3, 47], and the three modules (*Mapit*, *STS*, and *MBP*) in SwarmMap exactly enable each agent to acquire an optimized local map in time.

5.2.2 Map Updating Latency Comparison

We further examine the end-to-end latency of each agent from uploading map segments to eventually obtaining the optimized map from the edge node. To save space in the figure, we denote SwarmMap, CCM-SLAM, and Multi-UAV as *S*, *C*, and *M*, respectively. Fig. 8c shows the averaged latency on map uploading, optimizing, and downloading of each system in different number of agent settings. As seen, the total latency of SwarmMap is around 95ms and 105ms for 5 and 15 agents respectively, outperforming baselines by > 40% and 65%. The majority part of the latency reduction comes from the data uploading and downloading because *Mapit* reduces the amount of data transfers to a large extent. On the other hand,

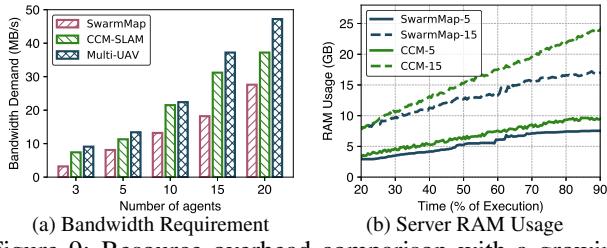


Figure 9: Resource overhead comparison with a growing number of agents.

the processing latency also drops around 10% when serving 15 agents as *STS* module reduces the averaged queuing delay.

5.2.3 Resource Overhead Comparison

Bandwidth Demand. We then measure the bandwidth demand of these three systems. As depicted in Fig. 9a, on average, SwarmMap reduces > 35%, 20%, 30%, 25%, 20% of network bandwidth requirement when serving 3, 5, 10, 15, 20 mobile agents compared with existing works. Said differently, SwarmMap could serve more agents with the same wireless link throughput. For instance, under 27.4MB/s shopping mall bandwidth limitation, SwarmMap can support more than 20 agents while existing works merely around 10.

RAM Usage. We stitch the 00-05 trajectories on the KITTI dataset to generate a trajectory with 16.2km length and conduct a 30min experiment to measure the RAM usage. As shown in Fig. 9b, compared to CCM-SLAM, SwarmMap saves an average memory overhead of 2GB and 6GB when serving 5 and 15 agents, respectively, and the map size becomes stable under an upper bound (as seen, 15GB when serving 15 agents) once the whole scene is well mapped. Unlike CCM-SLAM which requires transmission of a large volume of map elements, SwarmMap leverages *Mapit* and significantly reduces the bandwidth demand for map synchronization. In addition, the *MBP* module prunes the size of the global map maintained and optimized on the server, thus reducing the system overhead on computational resources.

Generally speaking, SwarmMap aims to scale the collaborative SLAM service with the same resource overhead at the edge. SwarmMap will achieve a better performance with more computational resources are allocated and advanced resource management technologies (e.g., swap or virtual memory) are leveraged on edge, which are left as future works.

5.3 Ablation Study

We then conduct an ablation study to understand the effectiveness of each module in SwarmMap.

Performance of *Mapit*. We compare *Mapit* with CarMap [2], CCM-SLAM [40], and benchmark (e.g., edgeSLAM [47] and Edge-SLAM [3] that directly transmit the entire map without feature compression). Table 1 records the average data interaction speed (i.e., the average amount of map data uploaded and downloaded by each agent per second) of them

Table 1: Transmitted data volume comparison.

Solution	Average Data Interaction Speed (MB/s)			
	TUM	KITTI	EuRoc	Shopping Mall
<i>Mapit</i>	1.3	1.1	1.3	1.4
CarMap	1.9	0.9	1.2	1.8
CCM-SLAM	3.2	1.9	2.2	2.9
Benchmark	5.2	4.3	4.7	4.9

Table 2: Map compression performance comparison.

Solution	KITTI 02		KITTI 05	
	Map Size (GB)	ATE (cm)	Map Size (GB)	ATE (cm)
<i>MBP</i>	3.1	7.6	1.9	6.4
Sum-Map	2.8	10.7	1.8	9.3
Benchmark	5.2	7.4	4.1	5.8

on different datasets. As seen, *Mapit* saves nearly two times the bandwidth compared to CCM-SLAM and benchmark on all datasets. *Mapit* performs slightly worse than CarMap on KITTI and EuRoc datasets, where the operating environments are relatively large (e.g., broad city roads). In these scenarios, the agents spend most of their time in the cold-start session during which they continuously transfer the newly generated map elements. In contrast, on TUM and our shopping mall datasets, the SLAM system completes the environment profiling quickly and soon enters the maintenance session during which *Mapit* eliminates map data transfer and saves the bandwidth by adopting the strategy of transmitting only records of map modifications rather than the modifications themselves.

Performance of *STS*. We evaluate *STS* by counting the average tracking lost percentage (i.e., proportion of video frames, with which agents fail to track themselves, in all video frames) of SwarmMap with (w/) and without (w/o) *STS*. As depicted in Fig. 10, despite the increasing service scale, SwarmMap (w/ *STS*) maintains a stable service quality, and the lost percentage is within 4% in all scenarios. In contrast, the lost percentage of CCM-SLAM as well as SwarmMap (w/o *STS*) increases rapidly, and the average lost percentage is at least 8% when serving more than 10 agents, which may lead to a terrible self-tracking and environmental mapping performance. Generally speaking, the *STS* strategy enables SwarmMap to prioritize tasks depending on the agent emergence states and prevent most agents from losing self-tracking.

Performance of *MBP*. We finally compare *MBP* with a map compression algorithm Sum-Map [27]. Specifically, we evaluate the map size after compression by their approaches and, equally important, the localization accuracy of each agent using the compressed map for self-tracking. The results are recorded in Table 2. We conduct experiments on the KITTI 02 and 05 trajectories because of the large map redundancy in them. The benchmark (only store the global map without compressing it) shows the size of the original map and the ATE by using it. As seen, *MBP* reduces the original map size by almost half. Although the map compression ratio of *MBP* is a little smaller than that of Sum-Map, *MBP* barely sacrifices the accuracy of the global map.

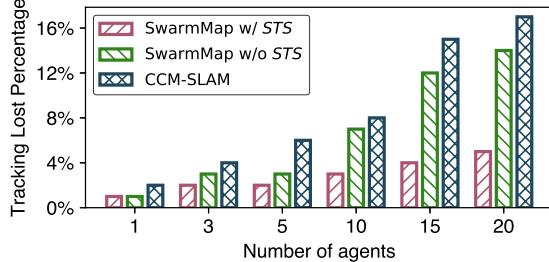


Figure 10: Tracking stability comparison.

6 Oil-field Case Study

Based on SwarmMap, we have developed a real-time collaborative visual SLAM system and deployed it in one of the world’s largest oil-field ($> 170km^2$) in the Middle East for industrial inspection. The details about the deployment setups can be found in §A.6. We conduct a three-month pilot study (from June 2021 to August 2021) and summarize our main findings regarding the SLAM accuracy and system overhead.

SLAM Accuracy. We calculate the average location error of each agent during inspections and present these results in Fig. 11 and Fig. 12. Note that we cannot directly obtain ground-truth in the same way as in the experiment (e.g., deploy expensive Lidar or Opti-Track cameras), hence we collect the video frames captured by all agents and run the multi-agent ORB-SLAM3 offline afterward without considering the system latency. On this basis, we take the difference between the real-time localization performance of SwarmMap and offline processed results as the location error. As shown in Fig. 11, the average location error is 19.3cm and 29.1cm in indoor and outdoor scenarios, respectively, satisfying the task requirement (1m and 1.5m for indoor and outdoor inspections). Fig. 12 further illustrates the performance of each agent, and we find that two outdoor inspection drones (agents 9 and 10) suffer from a higher location error (up to 1m). The reason behind it is that these two drones are carrying out oil pipeline inspection at the border of the oil field; they fly faster (e.g., $> 5m/s$) and far away from the edge server (e.g., 15km). Therefore, they may experience certain delays due to the data forwarding through multi-hop mesh networks. Such a transmission delay may set a barrier for the drones to obtain the optimized map in time, causing localization errors. Nevertheless, the worst localization error of these two drones still satisfies the localization requirement in the outdoor scenario.

Latency. We measure each agent’s onboard localization latency (the delay on estimating its own location from an input image) and map updating latency. The results are depicted in Fig. 13. We observe that each agent could localize itself in a real-time manner (i.e., the localization delay is within 35ms, typically the camera inter-frame interval). The average map updating delay is around 100ms. Although agent 9 suffers from a higher map updating delay (an average of 191ms) due to multi-hop data forwarding, it can still localize itself in real-time by leveraging its local map data.

Bandwidth demand. We record the total bandwidth demand

for indoor (4 agents) and outdoor (8 agents) inspection tasks. Fig. 14 shows a snapshot over a span of 175 minutes. We find there is a drop in bandwidth demand at 45min and 75min, respectively. This is because the SLAM system enters the maintenance session at these two time points. Thanks to *Mapit*, the transferred data volume in the maintenance session is significantly reduced, with 4MB/s for indoor and 11MB/s for outdoor inspections. Additionally, due to the relatively higher flight speed and map updating delay for outdoor drones, the edge server needs to frequently transmit updated maps to them in *Mapit pull* to prevent them from losing self-tracking, which results in the outdoor bandwidth demand fluctuates more dramatically than indoor ones.

RAM Usage. We further record the edge’s RAM usage when executing the indoor and outdoor inspection tasks. As shown in Fig. 15, the maximum RAM usage in the indoor and outdoor scenarios is around 20GB and 12GB, both of which are well below the capability (32GB) of the edge node.

On-board CPU Usage. We also record the CPU occupancy rate of SwarmMap task (mobile part) on agent 1 (indoor drone) and 6 (outdoor drone) and plot these results in Fig. 16. The CPU usage of the outdoor drone is in the range of 20%-35%, while the indoor drone is 22%-43% during the 210 minutes of inspections. Due to the high dynamics of the indoor environment, the agent has to frequently update the local map although the whole area is well-mapped, which takes up more CPU resources than outdoor environments. Note that SLAM is an underlying algorithm that provides an agent with location and environmental information, and SwarmMap still leaves more than 50% CPU computational resources for each agent to perform upper-layer applications (e.g., context-aware interaction, object detection, or segmentation).

7 Related work

We review the most related works in this section.

Visual SLAM. One of the most fundamental algorithms in robotics has been a topic of research in robotics and mobile systems for several decades [6]. It consists of the concurrent construction of a surrounding environment and the state estimation of the robot moving within it. Typically, systems use monocular cameras [15, 20], stereo cameras [29], or RGB-D cameras [31]. Some of the more well-known visual SLAM examples include RGBD-SLAM [16], RTAB-Map [21], and ORB-SLAM [7, 28, 29]. Although SwarmMap is implemented on the top of ORB-SLAM2 [29], it can be easily ported to other map point-based visual SLAM like S-PTAM [34]. Other multi-map merging or optimization algorithms leveraged in recent work like ORB-SLAM3 [7], can also be integrated into SwarmMap. Our platform can also be applied to some feature/map point-based multi-sensor SLAM systems like VI-ORB [30], VINS [35], mmWave SLAM [24, 44].

Edge-assisted Real-time SLAM. Recent studies [2, 3, 8, 23, 40, 47] speed up the computation-intensive tasks on agents by task partition and offloading workload to an edge server.

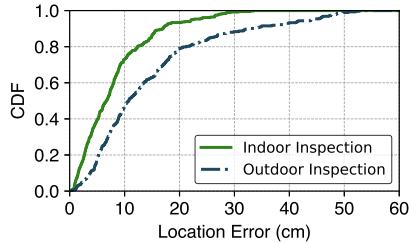


Figure 11: Different scenarios accuracy. Figure 12: Accuracy of different agents.

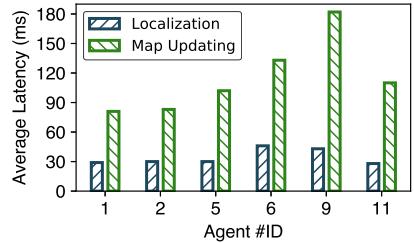
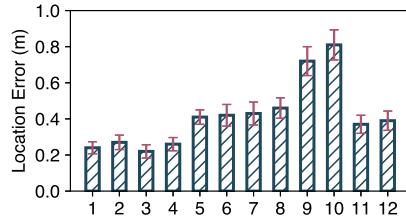


Figure 13: Latency measurement.

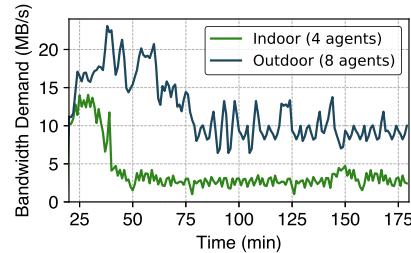


Figure 14: Bandwidth consumption.

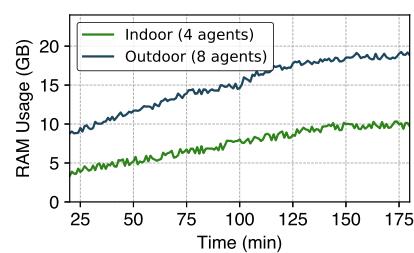


Figure 15: Edge server RAM usage.

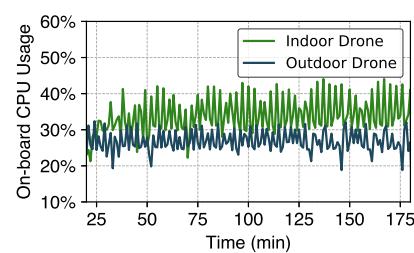


Figure 16: Mobile CPU occupation.

Therein, edgeSLAM [47] and Edge-SLAM [3] enable mobile agents to run visual SLAM in real-time. They split the original ORB-SLAM2 architecture and offload the *local mapping* and *loop closure* tasks to an edge server. CarMap [2] leverages the map constructed by crowdsourced agents and designs a near real-time map update framework between client and cloud. Muti-UAV [39] and CCM-SLAM [40] leverage a central server with potentially larger computational capacity to merge and optimize maps constructed by different agents, while each agent maintains partial local maps for tracking. However, as the number of serving agents scales, these works face severe scalability issues including excessive bandwidth consumption, severe localization errors, and large data storage. SwarmMap is the first work that solves these scalability issues based on the same edge settings.

Multi-agent Collaborative SLAM. Collaborative SLAM has been explored recently [6]. C2TAM [37], C-ORB [22], and CVI-SLAM [19] present collaborative SLAM frameworks based on PTAM [20], ORB-SLAM2 [29], and VI-ORB [7] respectively. CSfM [17] also proposes a framework to coordinate maps upload from different agents. In general, the system goals of these works and SwarmMap are orthogonal: above systems mainly focus on map fusion, optimization, and segmentation to generate a high-quality global map of the environment, ignoring the real-time performance of each agent and the entire system. In contrast, SwarmMap aims at solving the scalability issues and support each agent for real-time tracking, mapping, and map updating. Inspired by current efforts, we could integrate some map merging, optimizing, and even compressing algorithms proposed by recent works [6, 13, 14, 27, 49] into SwarmMap for a better SLAM performance, which are left as future works.

8 Discussion

We briefly discuss limitations and future work in this section.

The capacity of SwarmMap. Although SwarmMap signif-

icantly reduces the bandwidth consumption and memory overhead for collaborative visual SLAM systems, such resource consumption still grows linearly with the number of the agents, which still fundamentally limits the system capacity. The way to make the resource consumption grow sub-linearly [18] with respect to the number of agents worth further research. On the other hand, the current *Mapit* design merely focuses on reducing bandwidth consumption in the maintenance session. Serving the system throughput the entire life-cycle with *Mapit* could potentially save more bandwidth.

Map optimization algorithms integration. SwarmMap provides a basic map transmission and management platform for multi-agent SLAM. To date, SLAM map optimization is still a trending topic in the robotics field. Integrating existing advanced technologies (e.g., map compression, fusion, and semantic recognition) into SwarmMap for a better system performance is an ongoing work. Furthermore, efficient map data sharing not only between mobile and edge, but among different agents could also benefit upper layer applications.

9 Conclusions

We have presented the design and implementation SwarmMap, a framework to support real-time collaborative visual SLAM at edge devices. SwarmMap proposes functionality and resource abstractions of SLAM systems and provides three light-weight system services to address the communication, storage, and scheduling issues in edge-based scenarios. We implement SwarmMap as a software package on the ROS platform so that most variations of visual SLAM systems can directly benefit from it. Extensive evaluations and a three-month pilot study demonstrate its superior performance.

Acknowledgements

We thank the MobiSense group, the anonymous reviewers and our shepherd, Ramesh Govindan, for their insightful comments. This work is supported in part by the NSFC under grant 61832010, 61972131.

References

- [1] Numba GPU Acceleration. <https://numba.pydata.org/>.
- [2] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. Carmap: Fast 3d feature map updates for automobiles. In *Proceedings of the USENIX NSDI*, 2020.
- [3] Ali J Ben Ali, Zakiyah Sadat Hashemifar, and Karthik Dantu. Edge-slam: edge-assisted visual simultaneous localization and mapping. In *Proceedings of the ACM Mobicom*, 2020.
- [4] ArduPilot. <https://ardupilot.org/ardupilot/>.
- [5] Berta Bescos, José M Fácil, Javier Civera, and José Neira. Dynaslam: Tracking, mapping, and inpainting in dynamic scenes. *IEEE Robotics and Automation Letters*, 3(4):4076–4083, 2018.
- [6] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on robotics*, 2016.
- [7] Carlos Campos, Richard Elvira, Juan J Gómez Rodríguez, José MM Montiel, and Juan D Tardós. Orb-slam3: An accurate open-source library for visual, visual-inertial, and multimap slam. *IEEE Transactions on Robotics*, 2021.
- [8] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the ACM Sensys*, 2015.
- [9] EuRoC Dataset. <https://projects.asl.ethz.ch/datasets/>.
- [10] KITTI Dataset. http://www.cvlibs.net/datasets/kitti/eval_odometry.php.
- [11] TUM Dataset. <https://vision.in.tum.de/data/datasets/rgbd-dataset/tools>.
- [12] DJI drones for industrial inspection. <https://www.dji.com/products/industrial>.
- [13] M. Dymczyk, S. Lynen, M. Bosse, and R. Siegwart. Keep it brief: Scalable creation of compressed localization maps. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2536–2542, 2015.
- [14] Marcin Dymczyk, Thomas Schneider, Igor Gilitschenski, Roland Siegwart, and Elena Stumm. Erasing bad memories: Agent-side summarization for long-term mapping. In *Proceedings of the IEEE IROS*, 2016.
- [15] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *Proceedings of the Springer ECCV*, 2014.
- [16] Nikolas Engelhard, Felix Endres, Jürgen Hess, Jürgen Sturm, and Wolfram Burgard. Real-time 3d visual slam with a hand-held rgb-d camera. In *Proceedings of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum, Västerås, Sweden*, volume 180, pages 1–15, 2011.
- [17] Christian Forster, Simon Lynen, Laurent Kneip, and Davide Scaramuzza. Collaborative monocular slam with multiple micro aerial vehicles. In *Proceedings of the IEEE IROS*, 2013.
- [18] Samvit Jain, Xun Zhang, Yuhao Zhou, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Paramvir Bahl, and Joseph Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *Proceedings of the IEEE/ACM SEC*, 2020.
- [19] Marco Karrer, Patrik Schmuck, and Margarita Chli. Cv-slam—collaborative visual-inertial slam. *IEEE Robotics and Automation Letters*, 3(4):2762–2769, 2018.
- [20] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In *Proceedings of the IEEE ISMAR*, 2007.
- [21] Mathieu Labbé and François Michaud. Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 2019.
- [22] Fu Li, Shaowu Yang, Xiaodong Yi, and Xuejun Yang. Corb-slam: a collaborative visual slam system for multiple robots. In *International Conference on Collaborative Computing: Networking, Applications and Work-sharing*. Springer, 2017.
- [23] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *Proceedings of the ACM Mobicom*, 2019.
- [24] Chris Xiaoxuan Lu, Stefano Rosa, Peijun Zhao, Bing Wang, Changhao Chen, John A Stankovic, Niki Trigoni, and Andrew Markham. See through smoke: robust indoor mapping with low-cost mmwave radar. In *Proceedings of the ACM MobiSys*, 2020.
- [25] Yunfei Ma, Nicholas Selby, and Fadel Adib. Drone relays for battery-free networks. In *Proceedings of the ACM Sigcomm*, 2017.

- [26] ROS Melodic. <https://wiki.ros.org/melodic>.
- [27] Peter Mühlfellner, Mathias Bürki, Michael Bosse, Wojciech Derendarz, Roland Philppsen, and Paul Furgale. Summary maps for lifelong visual localization. *Journal of Field Robotics*, 33(5):561–590, 2016.
- [28] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.
- [29] Raul Mur-Artal and Juan D Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgbd cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.
- [30] Raúl Mur-Artal and Juan D Tardós. Visual-inertial monocular slam with map reuse. *IEEE Robotics and Automation Letters*, 2(2):796–803, 2017.
- [31] Richard A Newcombe, Steven J Lovegrove, and Andrew J Davison. Dtam: Dense tracking and mapping in real-time. In *Proceedings of the IEEE ICCV*, 2011.
- [32] Van Opdenbosch et al. *Data Compression for Collaborative Visual SLAM*. PhD thesis, Technische Universität München, 2019.
- [33] Opti-Track. <https://optitrack.com/>.
- [34] Taihú Pire, Thomas Fischer, Gastón Castro, Pablo De Cristóforis, Javier Civera, and Julio Jacobo Berllés. S-ptam: Stereo parallel tracking and mapping. *Robotics and Autonomous Systems*, 93:27–42, 2017.
- [35] Tong Qin, Peiliang Li, and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *Proceedings of the IEEE Transactions on Robotics*, 2018.
- [36] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. Avr: Augmented vehicular reality. In *Proceedings of the ACM MobiSys*, 2018.
- [37] Luis Riazuelo, Javier Civera, and JM Martinez Montiel. C2tam: A cloud framework for cooperative tracking and mapping. *Robotics and Autonomous Systems*, 62(4):401–413, 2014.
- [38] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the icp algorithm. In *Proceedings of the IEEE 3-D digital imaging and modeling*, 2001.
- [39] Patrik Schmuck and Margarita Chli. Multi-uav collaborative monocular slam. In *Proceedings of the IEEE ICRA*, 2017.
- [40] Patrik Schmuck and Margarita Chli. Ccm-slam: Robust and efficient centralized collaborative monocular simultaneous localization and mapping for robotic teams. *Journal of Field Robotics*, 36(4):763–781, 2019.
- [41] CUDA Toolkit. <https://developer.nvidia.com/cuda-toolkit>.
- [42] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle adjustment—a modern synthesis. In *Proceedings of the Springer International workshop on vision algorithms*, 1999.
- [43] Dominik Van Opdenbosch and Eckehard Steinbach. Collaborative visual slam using compressed feature exchange. *IEEE Robotics and Automation Letters*, 4(1):57–64, 2018.
- [44] Teng Wei, Anfu Zhou, and Xinyu Zhang. Facilitating robust 60 ghz network deployment by sensing ambient reflectors. In *Proceedings of the USENIX NSDI*, 2017.
- [45] Amazon Warehouse with Robots. <https://www.wired.com/story/amazon-warehouse-robots/>.
- [46] Nvidia Jetson AGX Xavier. <https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit>.
- [47] Jingao Xu, Hao Cao, Danyang Li, Kehong Huang, Chen Qian, Longfei Shangguan, and Zheng Yang. Edge assisted mobile semantic visual slam. In *Proceedings of the IEEE INFOCOM*, 2020.
- [48] Shengkai Zhang, Wei Wang, and Tao Jiang. Wi-fi-inertial indoor pose estimation for microaerial vehicles. *Transactions on Industrial Electronics*, 68(5):4331–4340, 2020.
- [49] Danping Zou, Ping Tan, and Wenxian Yu. Collaborative visual slam for multiple agents: A brief survey. *Virtual Reality & Intelligent Hardware*, 1(5):461–482, 2019.

A Appendix

A.1 Functions Registered in *Mapit*

In the *Mapit add* module, we dig the insights about how map elements get changed and find these changes mainly caused by certain important SLAM functions, a fraction of which is listed in Table 3. Thus, modifications that happened to the map can be recorded as calling history of these functions. For certain functions shown in the table, some removal and compression on the records will not harm data consistency. For instance, if a function is marked as overwritten, it indicates that its only effective change on a map element is the latest one i.e., changing the pose of a map point. As for those

Table 3: Functions that could change the map element (only some fundamental functions are listed)

Target	Function	Type	Description
KeyFrame	SetPose	overwritten	set the pose of the keyframe
KeyFrame	AddMapPoint	unique	add a map point to the keyframe
KeyFrame	EraseMapPointMatch	unique	remove a map point from the keyframe
KeyFrame	SetBadFlag	unique	mark the keyframe bad and delete it
MapPoint	SetWorldPos	overwritten	set map point position in the world coordinate
MapPoint	AddObservation	unique	add a keyframe that observes the map point
MapPoint	EraseObservation	unique	remove a keyframe from observations
MapPoint	SetBadFlag	unique	mark the map point bad and delete it
MapPoint	IncreaseVisible	stackable	increase the count that map point is observed
MapPoint	IncreaseFound	stackable	increase the count that map point is matched
MapPoint	SetLastTrackedTime	overwritten	set the last tracked time of the map point
MapPoint	UpdateNormalAndDepth	overwritten	update the normal vector and depth of the map point
Map	Clear	overwritten	clear the current map
Map	AddLoopClosing	unique	add a keyframe to loop closing queue

Table 4: Dataset Description

Dataset Label	Trajectory Sequence	Total Time (min)	Total Path (m)	Total Frames	Environment
T-M (TUM Medium & Easy)	fr2_desk	1.66	18.88	2965	office
	fr3_long_office_household	1.45	21.46	2585	
T-D (TUM Difficult)	fr2_large_with_loop	2.88	39.11	5182	industrial hall
	fr2_large_no_loop	1.87	10.93	3359	
K-M (KITTI Medium & Easy)	00 / 05	7.57 / 4.79	3724.18 / 2205.58	4541 / 2761	city road
K-D (KITTI Difficult)	02 / 04	7.77	5067.23 / 393.65	4661 / 271	city road
E-M (EuRoC Medium & Easy)	MH_01 / MH_02	2.47 / 2.50	68.52 / 73.50	3682 / 3040	machine hall
E-D (EuRoC Difficult)	MH_04 / MH_05	1.65 / 1.85	91.70 / 97.59	2033 / 2273	machine hall
Shopping Mall (Our Dataset)	N/A	15	314.2	24,365	shopping mall

marked stackable, the implication is that records about modifying the same element can be merged by parameters and still yield the same effect.

A.2 Map-point Evaluation Metrics

A typical SLAM map consists of two types of elements, map points and keyframes. Map points represent discrete 3D landmarks in the global coordinate, and keyframes are selected frames indicating poses and positions of the corresponding camera (as illustrated in Fig. 18 with corresponding notations in Table 5). EBM [14] introduces several features based on local geometry information; we list four important metrics to evaluate a map-point we used in *MBP*:

- **Observing Path Length.** The distance traveled while observing the map-point and is obtained as

$$\phi_d^i = \sum_{j \in S^i} \|\mathbf{t}_G^{j+1} - \mathbf{t}_G^j\|_2.$$

- **Maximum Observing Distance.** The distance traveled between two most distant keyframes on a track, and each of them observes the map-point. Its computation requires maximization over all keyframes observing the same map-point, i.e.,

$$\phi_\delta^i = \max_{j,k \in S^i} \|\mathbf{t}_G^j - \mathbf{t}_G^k\|_2.$$

- **Maximum Observing Angle.** The maximum angle between two keyframes that could observe the map-point and is

obtained as

$$\phi_a^i = \max_{j,k \in S^i} \arccos(\mathbf{r}_G^{j,i} \cdot \mathbf{r}_G^{k,i}).$$

- **Mean Re-projection Error.** Apart from the map-point track geometry, it is also worth considering the consistency of the map in the map-point’s locality. EBM calculate the average re-projection error of each map-point to represent the mapping stability, i.e.,

$$\phi_p^i = \frac{\sum_{j \in S^i} \|m_{i,j} - m'_{i,j}\|_2}{|S^i|}.$$

A.3 SwarmMap Abstraction

Fig. 17 shows the high-level abstraction of SwarmMap’s implementation. The *MBP* module assists the map fusion and optimization unit to eliminate the data redundancy in the global map. The *STS* module replaces those handcrafted request handlers in conventional SLAM implementations [19, 39, 40] and thus alleviates the end users’ development overhead. Finally, we replace the communication unit and map handlers with a unified *Mapit* module. Such a layered implementation decouples SwarmMap’s functional modules, allowing the end-users to turn on/off each module as they need. It also avoids the deeply embedded manual code changes (e.g., defining handlers) that again challenge the system scalability.

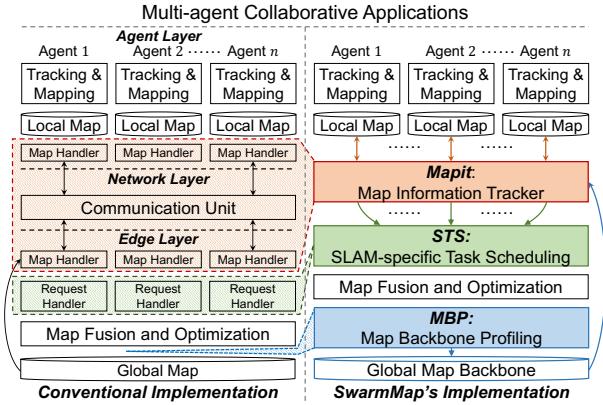


Figure 17: High-level abstraction of SwarmMap’s implementation. The arrow shows the data flow.

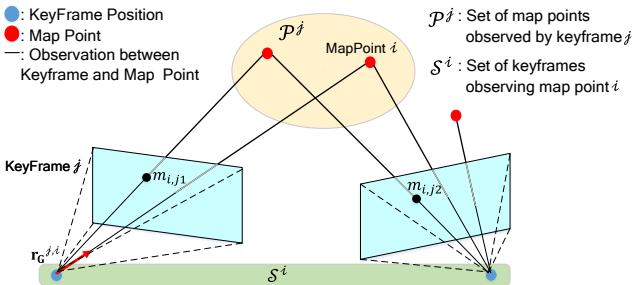


Figure 18: Observation connection between keyframes and map-points.

A.4 Experimental Dataset Description

We list the public datasets, the trajectories we used, and our handcrafted shopping mall dataset in Table 4. We select representative trajectories with different difficulty levels (in terms of environmental dynamics, path length, feature point sparsity, ambient light intensity, etc..) in TUM, KITTI, and EuRoc datasets, respectively.

A.5 Plug-and-play

We demonstrate the portability of SwarmMap by integrating each of its components into two different SLAM systems. We add *STS*, *Mapit*, and *MBP* to ORB-SLAM3 [7], the latest follow-up of the ORB-SLAM system, and measure the

Table 5: Notation Description

Notation	Description
\mathbf{x}_G^i	position of map point i in global coordinate G
\mathbf{t}_G^j	position of keyframe j in global coordinate G
S^i	set of all keyframes observing map point i
$\mathbf{r}_G^{j,i}$	unit-length observing vector starting from the observing keyframe j to map point i in global coordinate G
\mathcal{P}^j	set of all map points observed by keyframe j
\mathcal{M}^i	set of all agents observing map point i
t_c^i, t_l^i	creation and last tracked time for map point i

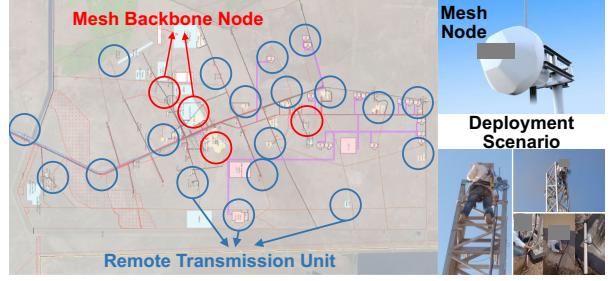


Figure 19: Mesh network deployment in the Oil-field.

accuracy gain brought by each module. Fig. 20 shows the results. As seen, all these three modules contribute to localization accuracy. When serving 5 agents, *STS*, *Mapit*, and *MBP* decrease location errors by around 50% and 30%, and 10%, respectively. The contribution of each component also grows with an increasing number of agents.

We also integrate SwarmMap into our baselines CCM-SLAM, Multi-UAV, and ORB-SLAM3 (abbreviated as *C*, *M*, and *O*, respectively) to explore the location error reduction. As depicted in Fig. 21, the location error of CCM-SLAM, Multi-UAV, and ORB-SLAM3 decreases by 13.4%, 12.2%, and 16.7% respectively in 5 agents settings. When serving 15 agents, the error decreases further to 17.2%, 31.3%, and 29.6%.

Remarks. These results show that most existing works in multi-agent scenarios (especially scenarios with more agents) can directly benefit from SwarmMap. It is worth mentioning that we do not re-design or modify the code structure of these existing works for integration. We merely provide a wrapper to hook up these systems and SwarmMap (i.e., call the API defined in SwarmMap).

A.6 Case Study Setups

Our system consists of 12 mobile agents to perform daily inspection tasks both indoors and outdoors. These agents communicate with an Nvidia Jetson AGX Xavier edge node through Wi-Fi mesh networks, as shown in Fig. 19.

Inspection agents. We have deployed 12 mobile agents to perform daily inspection tasks, including 4 DJI Inspire drones (Agent #ID 1-4, equipped with 2K cameras) for indoor warehouse inspection as well as 6 DJI Inspire (#ID 5-10) and 2 inspection vehicles (#ID 11-12, equipped with 1080P cameras) for outdoor oil-field inspection. For drones, we integrate the mobile part of SwarmMap into ArduPilot [4], a widely-used open source drone development platform. The output localization and mapping results are streamed to the ArduPilot Mega controller through a Micro-USB port for supporting upper-layer applications (e.g., real-time drone flight control, abnormal events detection). The two inspection vehicles are equipped with Nvidia Jetson TX1 as their computing units.

Edge server. We implement the edge side of SwarmMap on an Nvidia Jetson AGX Xavier edge node with a 32GB 256-Bit LPDDR4x RAM, a 16-core ARM v8.2 64-bit CPU, and a

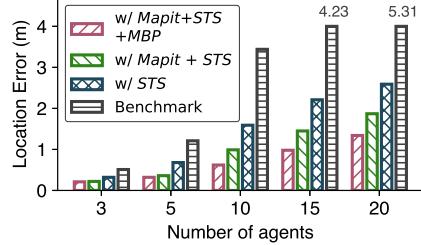


Figure 20: Performance of each module.

512-core Volta GPU. We also turn on the GPU acceleration by Numba [1] and CUDA [41] to speed up the back-end global map optimization procedure. The power consumption of the edge node is below 30W, which is less than the available power supply in the industrial scenario.

Wireless Network. The 4 indoor inspection drones communicate with the edge node via 2.4 GHz WiFi, while the 8 outdoor inspection agents communicate through a mesh net-

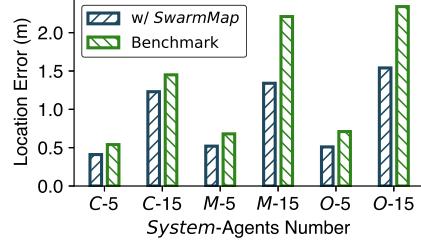


Figure 21: Performance gains.

work. In order to make the mesh network cover the whole 170km^2 outdoor oil-field (the west-east distance is around 30km), 24 communication nodes, including 4 mesh backbone nodes and associated 20 remote transmission units (RTU) are deployed (shown in Fig. 19). The maximum throughput measured by `iperf3` in the outdoor mesh and indoor WiFi network is 14.3MB/s and 26.8MB/s, respectively.

In-Network Velocity Control of Industrial Robot Arms

Sándor Laki¹, Csaba Györgyi¹, József Pető², Péter Vörös¹, and Géza Szabó³

¹*ELTE Eötvös Loránd University, Budapest, Hungary*

²*Budapest University of Technology and Economics, Budapest, Hungary*

³*Ericsson Research, Budapest, Hungary*

Abstract

In-network computing has emerged as a new computational paradigm made possible with the advent of programmable data planes. The benefits of moving computations traditionally performed by servers to the network have recently been demonstrated through different applications. In this paper, we argue that programmable data planes could be a key technology enabler of cloud and edge-cloud robotics, and in general could revitalize industrial networking. We propose an in-network approach for real-time robot control that separates delay sensitive tasks from high-level control processes. The proposed system offloads real-time velocity control of robot arms to P4-enabled programmable data planes and only keeps the high-level control and planning at the industrial controller. This separation allows the deployment of industrial control in non-real-time environments like virtual machines and service containers running in a remote cloud or an edge-computing infrastructure. In addition, we also demonstrate that our method can smoothly control 100s of robot arms with a single P4-switch, enables fast reroute between trajectories, solves the precise synchronization of multiple robots by design and supports the plug-and-play deployment of new robot devices in the industrial system, reducing both operational and management costs.

1 Introduction

In the recent decade, there has been an increasing demand from customers towards the manufacturing industry to provide more and more customized products. Personalized production is one of the key motivations for manufacturers to start leveraging new technologies that enable to increase, for instance, the flexibility of production lines. High flexibility, in general, is needed to realize cost-effective and customized production by supporting fast reconfiguration of production lines, as well as, easy application development. Fast reconfiguration and agile behavior can be achieved by moving the

robot control from the pre-programmed local robot controllers to the cloud. In industrial robotics research, cloud robotics is a major topic and in the last years, several studies [9, 11, 13] have shown the benefits of connecting robots to a centralized processing entity: a) usage of more powerful computing resources in a centralized cloud especially for solving Machine Learning (ML) tasks; b) lower cost per robot as functionalities are moved to a central cloud; c) easy integration of external sensor data and easier collaboration or interaction with other robots and machinery; e) reliability of functions can be improved by running multiple instances as a hot standby in the cloud and the operation can immediately be taken over from faulty primary function without interruption.

Though centralized processing has clear benefits in making the management of industrial processes simple and flexible, cloud-based solutions cannot satisfy the low latency and high reliability network requirements of real-time industrial control (e.g., velocity or torque control of actuators, robot arms, conveyor belts, etc.). Industry 4.0 and 5G propose the use of edge computing infrastructure for this purpose, moving these tasks to the computing nodes located close to the industrial environment. Though the propagation delay can significantly be reduced with this setup, edge-computing nodes rely on the same virtualization technologies as remote cloud infrastructures. Existing solutions require real-time operating systems to eliminate the effects of CPU scheduling and ensure precise timing (e.g., in velocity control the velocity vectors need to be sent to the robot arms with accurate timing). Newer robot arms have 2 ms or less update time. The real-time velocity control of hundreds of such robot arms requires an ultra-fast response time that is hard to satisfy with traditional edge computing infrastructure.

With the advent of PISA switches [3] and the P4 language [2], a new era has begun in which programmable network devices can not only perform pure packet forwarding but simple computations as well. This trend led to the birth of a new computational paradigm called in-network computing, where server-based computations or a part of them are moved to programmable data planes. This new way of using network-

Source code is available at <https://github.com/slaki/nsdi22>.

ing hardware can open up the fields for low-latency real-time calculations on the application level during the communication. Foremost, they can split long, distant control loops into smaller ones to deal with transport latency, enable computations at line rate and ensure real-time response time in orders of microseconds, solving the previously described problems of cloud and edge-cloud robotics.

In this paper, we investigate how cloud robotics can benefit from the advances of in-network computing. In particular, we propose a system in which high-level control of industrial processes can be deployed in the cloud (or edge cloud) while low-level speed control of the robot arms is offloaded to the programmable data plane (switch, smart NIC, or service card). Similarly to recent practical deployment options [6], we only assume reliable network connections with low latency between industrial robots and the programmable data plane. This design has the advantage that the high-level industrial controller does not require real-time OS and has less strict end-to-end delay requirements. Our vision is that P4-programmable data planes (e.g., smart NICs, service cards, switches) could complement the computational capabilities of cloud and edge cloud infrastructures for use cases where real-time operation, ultra-fast response time, high throughput, or all of these are required. Though the proposed method controls robot arms independently, we also demonstrate that it can easily synchronize the low level control processes of multiple robots and thus can potentially provide support for coordinated operation.

Moving low-level robot control to the network poses many challenges that are addressed in this paper: 1) How can velocity control be implemented with the limited instruction-set of programmable hardware data planes? 2) What is an efficient trajectory representation? 3) What to do if the entire trajectory does not fit into the memory? 4) How can match-action tables be used as playback buffers of trajectories? 5) How can trajectory segments be loaded in the limited memory of the switch and updated without violating timing requirements? 6) What constraints are needed for the data and control plane interactions? 7) How can the low-level control of multiple robot arms be synchronized? 8) How can switching to an alternative trajectory be solved in run-time (e.g., implementing a collision avoidance or emergency stop operations)?

2 Related Work

The related work of this paper covers a wide area of expertise from various research fields. We grouped them according to the different topics.

Traditional characteristics of robots. An industrial robot has many metrics and measurable characteristics, which will have a direct impact on the effectiveness of a robot during the execution of its tasks. The main measurable characteristics are repeatability and accuracy. In a nutshell, the repeatability of a robot might be defined as its ability to achieve repetition

of the same task. While, accuracy is the difference (i.e., the error) between the requested task and the realized task (i.e., the task actually achieved by the robot). For more details about the calculation of accuracy and repeatability, see [10]. The ultimate objective is to have both; a robot that can repeat its actions while hitting the target every time. When the current mass production assembly lines are designed, robots are deployed to repeat a limited set of tasks as accurately and as fast as possible to maximize productivity and minimize the number of faulty parts. The reprogramming of the robots rarely occurs, e.g., per week, per month basis and it takes a long time, e.g., days, requiring a lot of expertise.

Network aspects Authors of [7] compare the network protocols used nowadays in industry applications e.g., Modbus, Profinet, Ethercat. All investigated Industrial Ethernet (IE) systems show similar basic principles, which are solely implemented in different ways. Several solutions apply a shared memory and most systems require a master or a comparable management system, which controls the communication or has to be configured manually. Shared memory is implemented via data distribution mechanisms that are based on high frequency packet sending patterns. These packets have to be transmitted with strict delivery time and small jitter. IE protocols rely so heavily on the transport network that protocol mechanisms common in broadband usage like reliable transmission, error detection, etc. are not among the basic features of industrial protocols. Authors of [1] summarize the fundamental trade-offs in 5G considering various dimensions of block-lengths, spectral efficiency, latency, energy consumption, reliability, etc. Numerous aspects have to be solved during an industry automation task even when the robot stands still.

In-Network Industrial Control In-network control is a way to offload critical control tasks into network elements managed and organized through a remote environment. In the past few years, numerous papers offered solutions for In-Network Complex Event Processing (CEP). These works focus on sensor data-driven event triggering based on specific threshold values. Authors of [15] demonstrate such a system for a strongly delay-sensitive use case, controlling an inverted pendulum. By outsourcing the control to a distant controller, they show how a very low RTT of 5-20ms can break the entire system or make it oscillate badly. By combining in-network processing with the distant controller, they were able to utilize the ultra-low latency of local communication, and the control of the pendulum showed identical results as with fully local control. This paper mainly focuses on the implementation of the LQR controller in P4 and the limitations of the P4 language. Though the method we propose in this paper also uses a controller (PID-like) in the middle of the pipeline, it goes much further by providing an abstract representation of function components with error bounds that can potentially be used in any controller algorithms. In addition, our approach also handles many other problems: trajectory-based

control, switching between trajectories, synchronization of multiple robots, etc. In [12] authors demonstrate their own P4-based CEP rule specification language. P4CEP’s system model works with a collection of end-systems that are interconnected by programmable network processing elements. End-systems are differentiated into event sources, and event sinks where the sinks can react to certain conditions observed by the event sources. FastReact [20] is another In-Network CEP system that advocates the idea to outsource parts of an industrial controller logic to the data plane by making the programmable switches able to cache the history of sensor values in custom data structures, and trigger local control actions from the data plane. [4] shows a robot control system where a P4 switch is located between an emulated robot arm and the controller. The switch can analyze both sides of the traffic. If it detects that a position threshold is violated by the robot, it sends back an emergency stop message within a very short time due to the local communication. This work only covers this simple failure detection scenario and cannot deal with the more advanced control of robot arms we show in this paper.

3 System Design

The main goal of this paper is to demonstrate the feasibility and practical benefits of programmable data planes in low-level industrial control. To this end, we show how real-time velocity control of robot arms can be implemented in P4-programmable network devices and how they can be integrated into the existing industrial ecosystem. Fig. 1 depicts the high-level architecture of the proposed system, enclosing one or more robot arms, a P4-switch, and an industrial controller. It is important to note that this is a practical deployment option. The first phase of the introduction of wireless communication into production cells looks similar [6].

Robot arms. We assume simple robot arms without in-built intelligence. Each robot arm consists of a number of joints controlled by actuators (i.e., servo motors). The actuators work independently, stream their internal state (position and velocity) at a constant frequency (generally in the range of 100Hz-1kHz) and require velocity control messages at a pre-defined rate (generally 100Hz-500Hz) to keep the movement smooth. Note that lost command messages may cause lags in the movement or deviance from the desired path to be followed. In our system model, each robot arm is handled as a set of actuators controlled in sync. However, many complex industrial processes also require the synchronized operation of multiple robots (or other devices like conveyor belts, etc.). In the proposed system, this case can naturally be deduced to the single robot case by handling the cooperative robots as a single entity with all the actuators of the participating individual robots.

P4-switch. A programmable packet processing device supporting the P4 language [2] (e.g., PISA switch, smartNIC, or distributed service card) that processes the status streams of

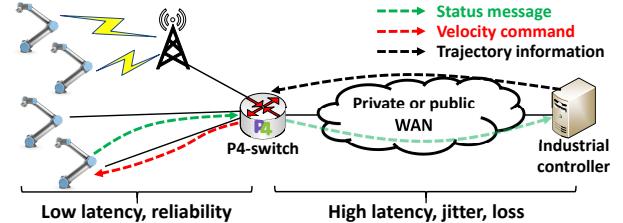


Figure 1: System overview.

the robot joints and generate the velocity control commands from the state messages and the desired trajectory provided by the industrial controller. We assume a highly reliable network connection with suitably small propagation delay (depending on the robot’s control frequency) between the P4-switch and the robots.

Industrial controller. It is responsible for coordinating the industrial processes at a high-level and thus planning the trajectories to be followed by the robot arms, re-planning trajectories if needed (e.g., for collision avoidance), verification of the process, failure detection and response, and synchronizing high-level processes. In our system design, the controller could be deployed at remote or edge cloud infrastructure. In the case of remote cloud deployment, the delay between the switch and the industrial controller could be in the order of 10-100ms with significant jitter. In both cases, the high-level industrial controller does not require real-time OS and thus can operate in a VM. Note that the industrial controller also gets the status information of the robots needed for tracking the whole industrial process, but cannot directly send commands to the actuators. Instead, it fills the match-action tables of the switch with a sequence of trajectory points needed for the P4-switch for controlling the robots at a low-level.

During operation, each robot arm executes the trajectory planned by the industrial controller. A trajectory is represented by a sequence of trajectory points (TPs), where each TP has a unique identifier and is associated with a relative timestamp (starting with 0) and the expected state (joint velocities and positions) of the robot arm at the given point of the operational timeline. Two consecutive TPs may be far from each other in both time and joint spaces. In the proposed method, the P4-programmable switch is responsible for the transition between the two TPs by continuously updating the joint velocities of the robot arm.

A trajectory example is depicted in Fig. 2. The initial trajectory plan on the top is a sequence of snapshots describing the robot states at discrete points of time. In the snapshot images, the orange arm illustrates the final configuration to be reached and the other denotes the desired state in the given TP. A robot state is described by two vectors representing the desired joint velocities and joint positions. Note that though the robot arm moves in the Cartesian space (as shown in the figure), the industrial controller maps the trajectory to the joint space (with

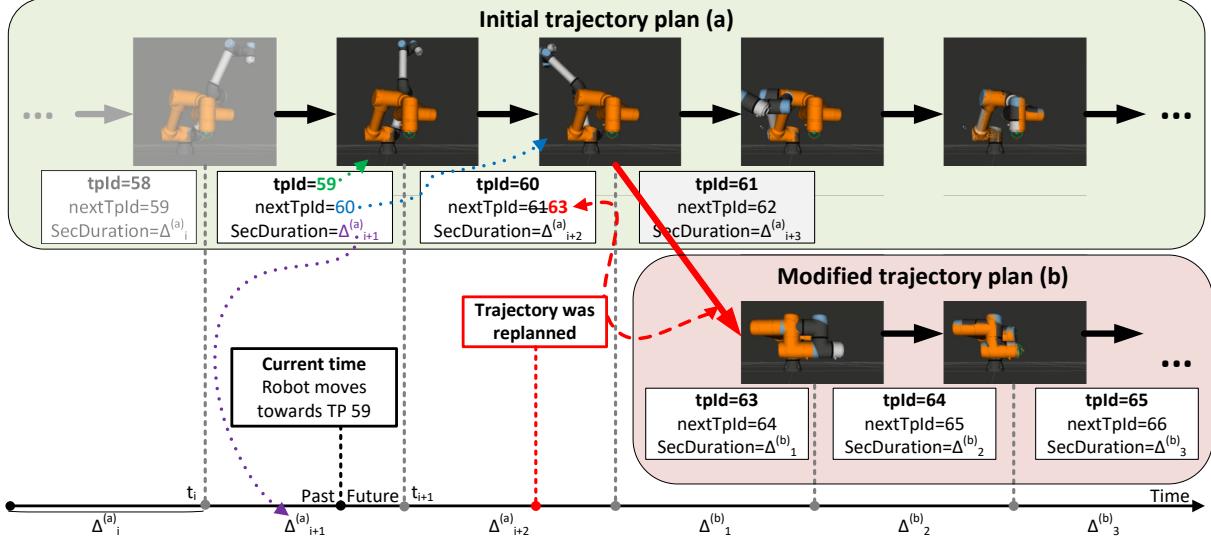


Figure 2: Trajectory example with re-planning.

units of rad/s and rad). One can also see that the transition from one TP to another needs to be performed in the allocated section duration of Δ_j . In the figure, the robot is heading TP 59. The new joint velocities to be set are calculated from the current state of the robot arm and the desired state in TP 59. Older TPs (e.g., 58 in the figure) have become obsolete. As soon as the target TP is reached, we switch to the next TP (60 in the example), heading the new associated robot state and also considering $\Delta_{i+2}^{(a)}$ dedicated for the transition (from TP 59 to 60).

3.1 System Requirements

In this section, we identify the minimal set of requirements needed for low-level real-time control of robot arms in most industrial use cases. We need to consider them during the implementation of the proposed system.

Velocity-control requirement. The smallest building blocks to be controlled are the actuators in our system. Actuators can be controlled independently. Each of them periodically generates status messages carrying the current joint velocity (rad/s) and joint position (rad) values. These messages first need to be parsed by the P4-switch responsible for low level control. Then, the switch has to calculate the new velocity value by applying feed-forward control (e.g., PID) that combines the state, timing, and trajectory information. Finally, the result shall be written into a command message and sent back to the actuator. Actuators are using different state reporting and command execution frequencies (generally the former is higher). Actuators operate at a given frequency. Each actuator first waits for a command message in a time window of constant length. If the time window is over, the actuator executes the command. If multiple commands are received in a time

window, only the latest is kept and all the others are dropped.

Timing requirement. The precise timing of control commands is crucial since the actuators of the robot arms expect incoming commands with a given frequency and do not tolerate large timeouts and jitter. In case of bursty arrival, a part of the commands may not be executed, leading to unexpected deviations from the desired trajectory. For example, an UR5e robot arm expects commands at 125 Hz, requiring a command message every 8ms. In addition, there are timing requirements between the P4-switch and the industrial controller on loading trajectory information. This requirement especially important if the entire trajectory cannot be stored in the switch (or it is not intended), and the controller periodically loads new TPs and deletes obsolete ones.

Synchronization requirement. Though we assume that actuators can be controlled separately, they are not independent. They belong to a single physical structure with its own kinematics. Thus, the actuators of a single robot need to be coupled in the control process. In addition, most industrial processes require the cooperation of multiple robot arms. Synchronization requirements can be defined on different time-scales. For example, if a robot stops in a position and then another process is started, but there is no strict time constraint (e.g., few seconds are acceptable) between the two processes, a remote industrial controller can even solve the synchronization. However, in several cases, this light synchronization is not enough, and thus the low-level control processes also need to work in sync (on a millisecond or sub-millisecond scale).

Trajectory switching requirement. The industrial controller continuously monitors the whole industrial process, and intervenes if needed (e.g., in case of failure or simple reconfiguration, or for collision avoidance purposes). This case is illustrated by Fig. 2 where the trajectory is modified (the red

point on the timeline), resulting in that after TP 60 the robot moves towards TP 63 instead of 61. Trajectory switching is needed when a robot is reconfigured or when an obstacle appears in the robot cell and collision avoidance can be ensured by the new trajectory.

Communication requirement. To reduce packet processing overhead in the P4-switch, we assume that robot arms apply a datagram-based communication protocol (e.g., native Ethernet frames, UDP packets, etc.) for sending status information and receiving commands. Both status and command messages consist of simple decimal fields in a binary format that can be parsed by the P4-switch with ease.

4 Robot Arm & Network Protocol

The robot arm used in our experiments is an UR5e [17]. UR5e is a lightweight, adaptable collaborative industrial robot with six joints (6-DOF). It is commonly used in research as it has a programmable interface, can be remotely controlled, provides industrial grade precision, and can operate alongside humans with no safeguarding. The robot vendor also provides a real-time emulation environment (URSim) that is fully compliant with the real robot arms and thus can be used for testing validation purposes.

UR5e can receive external commands described in UrScript [19] language via its network interface. It communicates with external controllers over TCP by default. However, the network protocol can be customized by adding a URCap [18] plugin (called daemon) to the robot. To make the communication simple and stateless, we created a URCap daemon implementing the translation between the original TCP-based and our UDP-based protocols.

During the protocol design we considered two practical aspects: 1) P4 capable devices are not suited for deep packet inspection, and thus cannot parse the entire content of large packets. It implies that every important field used for robot control has to be close enough to the beginning of the packet. 2) Both status and command messages of the original communication interface rely on floating point fields. However, the P4 language does not support floating-point arithmetic. This problem can be handled by multiplying each floating-point value with a properly large constant and then using the standard decimal operations. Though it is possible to implement this conversion in P4, it is much simpler and comfortable if the value is already in a decimal format in the used protocol.

Considering the above aspects, we use the same header structure for status and command messages encapsulated into simple IP/UDP packets. The introduced robot header (*rh*) consists of four fields: 1) a **robot ID** (*rh.RId*) used as a unique identifier of the robot arm, 2) a **joint ID** (*rh.JointId*) which determines the joint (or in general the actuator) of the given robot, 3) a **joint velocity** (*rh.velocity*) expressing the current speed (in rad/s) of the given joint in the status messages or the new joint-speed value to be set in the commands, and 4) a

joint position (*rh.position*) which is the current position (in rad) of the given joint in the status messages, and unset in the commands.

5 Velocity Control in Data Plane

Though our prototype is implemented in P4-16 with the Tofino Native Architecture (TNA), we aim at keeping the data plane description in this section general. In our model, the switch consists of two packet processing pipelines: an ingress and an egress. The two parts have different roles and responsibilities in the proposed implementation:

- **Ingress pipeline:** This part is responsible for 1) determining the current TP for the robot arm the status packet is sent by, 2) stepping the current TP to the next TP along the trajectory if required, or 3) switching to another trajectory in case of re-planning.
- **Egress pipeline:** This block solves the low-level velocity control by calculating the new joint velocity value based on the available information (state packet and trajectory).

5.1 Ingress pipeline

We assume that each TP can be identified by a unique ID. The memory layout of the ingress pipeline is depicted in Fig. 3. One can observe that we maintain three registers for each robot to be controlled. They store the identifiers of the current (REG_{tp}), the next (REG_{nextTp}) TPs, and the absolute timestamp ($REG_{nextTime}$) when the control has to step along the trajectory to the next TP. Fig. 2 provides a good illustration of the role of these three values. Accordingly, the robot moves towards the current TP (59) which should be reached at t_{i+1} ($REG_{nextTime}$) when we step forward to the next TP (60).

The ingress pipeline also contains two tables for storing the trajectory as a sequence of TPs and branching points where we can switch to another trajectory. Table `TPStepper` represents the trajectory to be followed by a robot arm as a linked list of TP identifiers. For each TP p , it stores the duration needed for moving from the previous TP to p and the identifier of the next TP that follows p along the trajectory. One can observe that the next TP determines how the robot arm continues its operation after reaching p .

The current and next TPs usually belong to the same trajectory, but in some cases, re-planning is required. Table `TrajectorySwitcher` solves this problem by switching between two trajectories. If there is a TP p along the original trajectory which could also be the starting point of the new trajectory, the switch can be implemented by replacing the next TP of p with the appropriate TP along the new trajectory. Thus after the branching point p , the robot arm starts following the new trajectory also loaded into table `TPStepper`.

Let us consider the example in Fig. 3 (see Fig. 2 for illustration). Table `TPStepper` is applied at time t_i when TP 59 becomes the new TP ($m.tpId = 59$) the robot arm is heading towards. At this point of time, the next TP is unknown and is filled from the table. The table also provides the section duration ($m.secDuration = \Delta_{i+1}^{(a)}$) allocated for reaching TP 59. This information is used for determining the absolute timestamp ($t_i + \Delta_{i+1}^{(a)}$) when the current TP is replaced by the next one (TP 60) and then table `TPStepper` is applied again. Though it sets REG_{nextTp} to 61 (next TP along the initial trajectory), table `TrajectorySwitcher` overwrites it with 63, the starting point of the new trajectory.

Algorithm 1 describes the ingress pipeline at a high abstraction level. At arrival, the status message from a robot executes the program block starting with line 3. First, the trajectory state ($TpId$, $nextTpId$ and the $nextTime$) is read from the registers. $TpId$ denotes the current TP the robot is currently heading towards and $nextTpId$ identifies the next TP. Then table `TrajectorySwitcher` is applied that replaces the $nextTpId$ if the current TP is a branching point. In most cases, there is no hit in this table. In line 6, we check if $nextTime$ is reached. If this condition is true, further actions (see line 11-16) are needed since we have to move to the next TP, update states (table `TPStepper`) and write them into the registers. In high-performance hardware data planes like Barefoot Tofino, registers can only be accessed once during the pipeline to ensure line-rate performance even at the Tbps scale. This constraint can be resolved by resubmitting the packet (lines 8-9). In this case, the ingress pipeline is executed twice only. Though packet resubmission can reduce the overall throughput, in practice this step is only performed when the current TP is reached. Note that in software targets the proposed pipeline could be implemented without the need for resubmission, but in turn, we can expect higher latency and performance limitations.

The proposed implementation has further practical benefits. In case of repetitive tasks which is usual in industrial scenarios, we can simply create loops in table `TPStepper` by setting the next TP to a TP visited previously. The synchronization of different robot arms can be solved either by merging the multiple robot arms into a single entity whose TPs represent the joint states of all participating robots or by creating a self-loop at the starting point of trajectories to be synchronized. In the latter case, if the section duration is long enough for inserting branching points to trajectories to be executed into table `TrajectorySwitcher`, the internal clock of the P4-switch ensures that robot arms start operating at the same time and are kept in sync during the industrial process.

5.2 Egress pipeline

The egress pipeline is responsible for calculating the velocity value to be set from the current state of the robot joint and the current TP ($tpId$). The new velocity value is computed

Algorithm 1: Ingress pipeline (pseudo-code)

```

Robot header: rh, Metadata: m;
Registers: REGtp, REGnextTp, REGnextTime;
Tables: TrajectorySwitcher, TPStepper;
apply block

1   if rh.isValid() then
2       if m.resubmitted==0 then
3           m.tpId = REGtp(rh.RId);
4           m.nextTpId = REGnextTp(rh.RId);
5           m.nextTime = REGnextTime(rh.RId);
6           TrajectorySwitcher.apply();
7           if m.nextTime>now() then
8               m.resubmit-needed = 1;
9               m.resubmit-data = m.nextTpId;
10      else
11          m.tpId = m.resubmit-data;
12          TPStepper.apply();
13          REGtp(rh.RId) = m.tpId;
14          REGnextTp(rh.RId) = m.nextTpId;
15          REGnextTime(rh.RId) += m.secDuration;
16          send_back();
17      else
18          Handling normal traffic (e.g., l2 forwarding);

```

by a simple PID-like controller, as the weighted sum of three values:

$$v_{new} = v_{curr} + c_1(v_{trg} - v_{curr}) + c_2(p_{trg} - p_{curr}),$$

where c_i s are constants, v_{curr} and p_{curr} denote the current speed and position of the robot joint while v_{trg} and p_{trg} are the desired joint velocity and position in the current TP. One can observe that the new velocity can be composed of three linear transformations: $(1 - c_1)v_{curr}, c_1v_{trg}, c_2p_{diff}$, where $p_{diff} = p_{trg} - p_{curr}$. Each actuator may have different physical properties and thus require different c_i constants in the transformations. The three `Transform` tables in Fig. 4 are used for approximating these linear transformations.

The egress control block is described in Algorithm 2. We first apply table `TargetData` to obtain the desired joint speed and joint position in the current TP ($m.tpId$). The actual state of the robot joint is carried by the robot header (rh). Lines 3-7 perform the primitive calculations needed for the P-controller mentioned previously. The new velocity is calculated as a sum of different components. Each component is calculated from metadata fields (diffPos stores the position difference) filled previously or from header fields by a transformation. The transformations are approximated by ternary or longest-prefix match (LPM) tables filled in run-time (see Sec. 5.3). If the calculated velocity value is too large, it can cause damage to the robot arm. To take the physical limits of the robot joints into account we introduce the table `LimitVelocity` checking

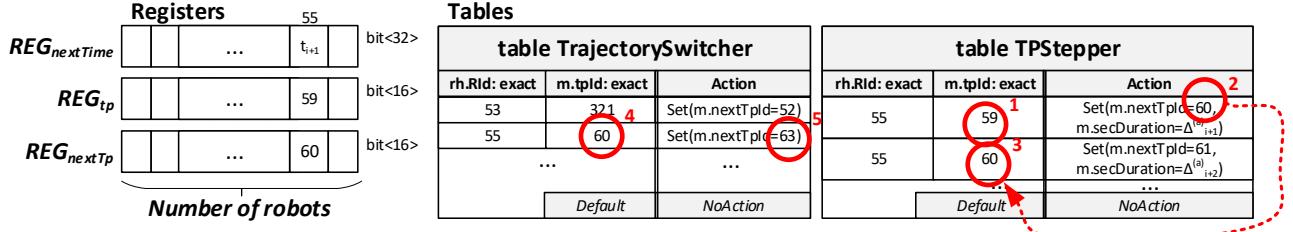


Figure 3: Memory layout at ingress.

whether the calculated velocity value is outside of the safety range, and mapping it into the normal range if needed. The calculated joint speed is encoded into the velocity field of robot header rh and sent back to the robot as a command message (lines 9-11).

Algorithm 2: Egress pipeline (pseudo-code)

```

Robot header: rh, Metadata: m;
Registers: -;
Tables: TargetData, LimitVelocity,
          TransformTrgVelocity, TransformCurrVelocity,
          TransformDiffPosition;

apply block
1   if rh.isValid() then
2     TargetData.apply();
3     m.diffPos = m.trgPos - rh.position;
4     TransformTrgVelocity.apply();
5     TransformCurrVelocity.apply();
6     TransformDiffPosition.apply();
7     rh.velocity += m.trgVel + m.diffPos;
8     LimitVelocity.apply();
9     swap_ipAddresses();
10    swap_udpPorts();
11    clear_checksums();
12  else
13    Handling normal traffic;

```

5.3 Approximating transformations

The new velocity value is calculated by applying transformations on some header or metadata fields. In our proof-of-concept P-controller, these transformations are simple multiplications with predefined constants, but this design enables us to apply even non-linear mappings.

Such a transformation can be approximated by a Longest-Prefix-Match (LPM) or a ternary-match table as depicted in Fig. 4. The match key is the parameter of the function (e.g., a header or metadata field), considering the most significant n bits starting with 1 (positive case) or 0 (negative case), as illustrated in Fig. 5. The action parameter is the function value

calculated from the significant bits only. Since we only use simple weight functions in our implementation, the relative error of the approximated output equals the relative error of the input, more precisely the relative error of the estimation based on the most significant n bytes. The estimated value for the input can vary between the largest and smallest possible values with the given prefix. During this process, we skip the leading zeros (or ones in case of negative values) and ignore the last k bits. Depending on this estimation, the relative error is less than or equal to $1/2^{n-1}$.

This approach fits well with the velocity control use case. If the input is small – suggesting that we are close to the target TP, we need to make a more precise movement – the approximation has a small absolute error. If the input has a higher absolute value – meaning that we are far from the target value, and high precision control is not needed – the method provides an acceptable higher absolute error.

This method can be improved with a small trick. The number of possible outputs is exactly the number of ternary entries. However, we can calculate the approximated value of $(c - 1)x$ instead of cx and add one more x to the result in the P4 program. This technique applied in Table TransformDiffPosition helped to improve the stability of the applied P-controller.

5.4 Limiting joint velocities

The different joints have their own physical properties that determine the maximum applicable velocity. To check the speed constraints and limit the velocity if needed, we apply table LimitVelocity. Let x be the velocity value ($rh.velocity$) to be tested and c be a constant value. Starting with the positive case, we can always decide whether $x > c$ if x has the same n long prefix as c but $x[n+1] = 1$ and $c[n+1] = 0$. Note that $x[1]$ denotes the most significant bit of x . The negative case is similar. If x has the same n long prefix as c but $x[n+1] = 0$ and $c[n+1] = 1$ then $x < c$. For an input of k bits, we need at most k entries in the table for each constant check. Fig. 4 shows a small example with the necessary prefix checks, considering a 16-bit long input value and predefined constant c . In the case of signed inputs, the first bit shall be handled carefully, but comparing to a negative number can be done similarly.

Tables

table LimitVelocity			
rh.Rid: exact	rhJointId: exact	rh.velocity: lpm	Action
53	0	0b 1/1	rh.velocity = c
53	0	0b 01/2	rh.velocity = c
53	0	0b 001/3	rh.velocity = c
53	0	0b 0001/4	rh.velocity = c
53	0	0b 000011111/9	rh.velocity = c
53	0	0b 0000111101/10	rh.velocity = c
53	0	0b 000011110111/12	rh.velocity = c
53	0	0b 000011110110 1/13	rh.velocity = c
53	0	0b 000011110110 011/15	rh.velocity = c
53	0	0b 000011110110 0101/16	rh.velocity = c
...		Default	
rh.velocity = c if rh.velocity > c (where c = 0b 0000 1111 0110 0100)		NoAction	

table TransformTrgVelocity			
m.trgVel: lpm	Action	m.trgVel =	
0b 10000..0/4	f(0b 1000 0..0)		
0b 10010..0/4	f(0b 1001 0..0)		
0b 10100..0/4	f(0b 1010 0..0)		
0b 10110..0/4	f(0b 1011 0..0)		
0b 11000..0/4	f(0b 1100 0..0)		
0b 11010..0/4	f(0b 1101 0..0)		
0b 11100..0/4	f(0b 1111 0..0)		
0b 01000..0/5	f(0b 0100 ..0..0)		
0b 01010..0/5	f(0b 0101 0..0..0)		
0b 010110..0/5	f(0b 0101 10..0..0)		
...		Default	
rh.velocity = f(x) + g(y) + h(z)		NoAction	

table TargetData			
rh.Rid: exact	m.tpid: exact	rhJointId: exact	Action
55	59	2	Set(m.trgPos=120, m.trgVel=1123)
55	60	2	Set(m.trgPos=180, m.trgVel=123)
...		Default	
...		NoAction	

table TransformCurrVelocity			
rh.velocity: lpm	Action	rh.velocity =	
0b 10000..0/4	g(0b 1000 0..0)		
0b 10010..0/4	g(0b 1001 0..0)		
0b 10100..0/4	g(0b 1010 0..0)		
0b 10110..0/4	g(0b 1011 0..0)		
0b 11000..0/4	g(0b 1100 0..0)		
0b 11010..0/4	g(0b 1101 0..0)		
0b 11100..0/4	g(0b 1111 0..0)		
0b 01000..0/5	g(0b 0100 ..0..0)		
0b 01010..0/5	g(0b 0101 0..0..0)		
0b 010110..0/5	g(0b 0101 10..0..0)		
...		Default	
rh.velocity = f(x) + g(y) + h(z)		NoAction	

table TransformDiffPosition			
m.diffPos: lpm	Action	m.diffPos +=	
0b 10000..0/4	h(0b 1000 0..0)		
0b 10010..0/4	h(0b 1001 0..0)		
0b 10100..0/4	h(0b 1010 0..0)		
0b 10110..0/4	h(0b 1011 0..0)		
0b 11000..0/4	h(0b 1100 0..0)		
0b 11010..0/4	h(0b 1101 0..0)		
0b 11100..0/4	h(0b 1111 0..0)		
0b 01000..0/5	h(0b 0100 ..0..0)		
0b 01010..0/5	h(0b 0101 0..0..0)		
0b 010110..0/5	h(0b 0101 10..0..0)		
...		Default	
rh.velocity = f(x) + g(y) + h(z)		NoAction	

Figure 4: Memory layout at egress.

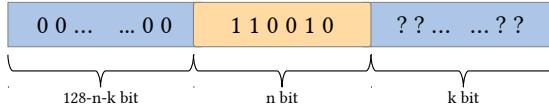


Figure 5: Considering the most significant n bits starting with 1 (positive case).

6 ROS integration

In this section, we briefly introduce our industrial controller implementation based on the Robot Operating System (ROS) [14] and its MoveIt [5] library used for generating and executing trajectories in a robot agnostic manner. ROS is an open-source robotics framework used in various robotics-related research since it can easily be extended and customized for specific use cases. Our ROS-based industrial controller uses MoveIt for motion planning and mobile manipulation of robots. In the proposed system, it generates a *JointTrajectory* message containing an array of points (timestamps, 6 joint positions, 6 joint velocities), as UR5e has six joints (as shown in Fig. 2).

The architecture of the industrial controller is shown in Fig. 6. The components developed to support the proposed system are marked by gray. They have been integrated with the standard MoveIt architecture consisting of trajectory generation using MoveIt planning, trajectory execution with MoveIt using standard ROS interface, and communication via the ROS driver of the UR5e arm.

To generate trajectories we can use RVIZ, a ROS visualizer software, with a MoveIt Motion Planning Plugin. Using RVIZ, we can generate trajectories interactively from a start point to a selected endpoint. Another way to generate trajectories is by 1) creating waypoints in Cartesian space, 2) then sending those points to a ROS node, 3) it computes a trajectory in joint space (defined by the joint angles of the robot) and 4)

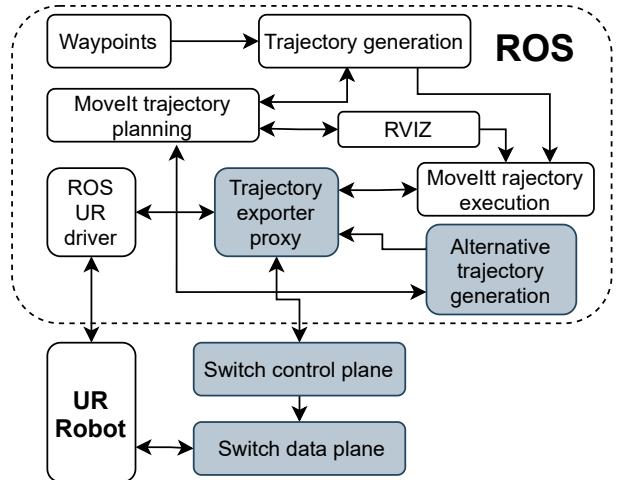


Figure 6: Trajectory generation and execution

finally it visits them in order.

The resulting joint trajectory is sent to MoveIt trajectory execution, which uses the ROS action interface defined by the UR driver to execute the trajectory on URSim/UR5e or it can also send the trajectory to the P4-switch's control plane via the Trajectory Exporter proxy, which fills the appropriate tables and let the switch execute the trajectory instead of the ROS UR driver.

6.1 Alternate trajectory generation

We developed Alternate trajectory generation to extend the existing capabilities of the system. Alternate trajectory generation and execution is a feature that leverages the ability of the P4 system to quickly change chains of trajectories, to execute prepared alternate trajectories in response to external changes e.g., in the robot's surroundings.

The alternate trajectory generation node uses MoveIt's tra-

jectory planning to prepare multiple branching trajectory fragments, then concatenates them into a single trajectory. The alternate trajectories are placed after each other, therefore the timestamps of the whole branching trajectory are not strictly incremental. Fig. 7 shows this process, the numbers indicate the timestamps of the trajectory’s start and endpoints.

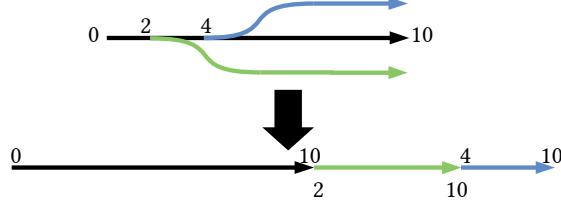


Figure 7: Generating a single trajectory from alternate ones

As the timestamps do not strictly increase, we can not use the trajectory execution of MoveIt. MoveIt is not able to execute alternate trajectories. Therefore we send the encoded trajectory to a proxy ROS component (node) which forwards the trajectory to the P4-switch.

We are also able to generate alternate trajectories on the fly when a trajectory is already being executed. We achieve this by keeping track of what is the current time in the currently executed trajectory. As we know the current time, we know where the robot will be in a Δt time. That point can be the start point of an alternate trajectory. To ensure a smooth transition during the switch between alternate trajectories we need to estimate the position of the switching as accurately as possible. To do this we need to estimate the 1) the latency between the industrial controller and the switch (RTT), and 2) the expected trajectory generation time ($t_{processing}$). For the estimation of the start position ($p_{start}(t)$) at time (t) we came up with the following formulae:

$$\begin{aligned} \Delta t &= t_{processing} + RTT \\ p_{predict}(t) &= p_{traj}(t + \Delta t) \\ &\quad + p_{status}(t) - p_{traj}(t) \\ &\quad + (v_{status}(t) - v_{traj}(t)) \times \Delta t \\ p_{start}(t) &= \lfloor p_{predict}(t) \times \frac{1}{g(t)} \rfloor * g(t) \end{aligned}$$

Where the error is estimated on the trajectory calculation side by calculating the difference of the position of the joints received in the last status message and the executed trajectory position. This is further adjusted by the difference of the current velocity of the joints and trajectory velocity times Δt . A binning of the values with an integer division and multiplication with the original granularity ($g(t)$) is applied on the predicted position value to replicate the behavior of the ternary table on the trajectory planner side and consider the granularity of the number representation in the specific time. $g(t)$ can be derived from the maximum of relative error (M_{rel} , see Sec. 5.3) by $g(t) = lpm(p_{traj}(t), M_{rel})$.

Fig. 8 shows an example on the error of $p_{start}(t)$ compared to $p_{status}(t + \Delta t)$, which is the joint position at the time of switching.

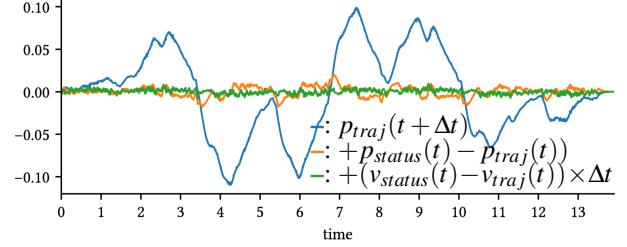


Figure 8: An example on the error of $p_{start}(t)$ compared to the later joint state

7 Evaluation

We carried out several experiments analyzing whether the proposed implementation can hold the identified system requirements. To this end, we deployed a simple testbed consisting of two servers (AMD Ryzen Threadripper 1900X 8C/16T 3.8 GHz, 128 GB RAM) and a Barefoot Tofino-based switch (STORDIS BF2556X-1T). One of the servers (called Server-A) is equipped with a dual port 10 Gbps NIC (Intel 82599ES) that supports hardware-based timestamping. This node is connected to the switch via two 10 Gbps links. The other server (Server-B) is equipped with a Mellanox ConnectX-5 dual port NIC whose ports are connected to the switch via two 100 Gbps links. For latency experiments, we used MoonGen tool [8] on Server-A with hardware-based timestamping to generate robot and mixed traffic. During the latency measurements, Server-B continuously generated non-robot background traffic with IP/TCP packets of size 1280B. We aimed to demonstrate the case that some ports of the P4-switch are dedicated to handling robot traffic while others forward normal traffic in parallel. For operational experiments, we used a real UR5e robot arm connected to the switch and Server-B was running one emulated UR5e [17] robot using the official URSim robot emulator. Note that the emulator provided by the robot vendor is fully realistic and works in real-time. During the experiments, we did not realize notable differences between the emulated and the real robot arm. In this scenario, our ROS-based industrial controller was run on Server-A and communicated with the control plane of the switch, loading and removing trajectory points.

The performed evaluation scenarios were designed to assess the proposed system in various common robotic use cases: 1) Pick and place actions: See Sec. 7.3, 2) Welding, painting, gluing: See Sec. 7.4, 3) Robot to robot collaboration: See Sec. 7.2, 4) Heterogeneous sensor and actuator deployment in the robot cell: See Sec. 7.1, 5) Agile control, safety, robot to human collaboration: See Sec. 7.5. The estimation

about scalability is covered by Appx. A.

7.1 Response time analysis and traffic load

A robot cell usually contains various sensor and actuator elements, meaning that there are other traffic sources in the robot cell than the one generated by the robot arm itself. The Supervisory Control And Data Acquisition (SCADA) systems also generate considerate background traffic. This is why it is important to evaluate the proposed system on a heterogeneous traffic mix. To this end, we carried out latency measurements under various traffic loads. In this scenario, MoonGen (Server-A) sent latency probes in every ms, mixed with various background traffic. The latency probes were valid robot status messages and thus they went through the entire robot control pipeline.

We evaluated the system with two different background traffic: 1) **Simple IP packets of 64B and 1280B sizes** were generated at variable sending rates (1-10 Gbps). The switch applied simple port forwarding and only the latency probes went through the robot-control pipeline. With small packet sizes the observed response time of robot traffic was in the range of [0.6 μ s, 1.3 μ s]. With packet size of 1280B the response time shifted towards 2 μ s as the load increased. This phenomenon was caused by one or more large background packets wedging between two latency probes. Note that at 10 Gbps transmitting a packet of size 1280B takes approx. 1 μ s. We also compared these measurements to the latency of a simple port forwarding program, the differences were not significant (<0.2 μ s). 2) **Robot status messages** were generated as background traffic, and thus all the packets went through the entire robot control pipeline. The latency results were basically identical with the previously described case of using 64B IP packets. The response time was ranging between 0.6 μ s and 1.3 μ s.

Though these measurements only show the response time in under-loaded situations without queueing effect, they are represented in most industrial environments where a number of assumptions can be made: 1) **predictable and stable load** since the device settings determine the packet generation frequencies; each device operates as a constant bit-rate source. The packet sizes are known and thus the overall load can easily be predicted. For example, a 6-DoF robot operating at 500 Hz (e.g., UR5e sends status messages at this rate; sending in every 2ms) generates approx. 1.5 Mbps status traffic on the upstream direction. Thus, the packet processing pipeline is required to ensure non-blocking operation at 3000 packets/s for a single robot. Considering 1000 robot arms which is far above the number of robots used in industrial setups nowadays, the required forwarding rate is 3M packets/s (approx. 1.5 Gbps) on average. However, considering synchronized robots whose status messages are sent within a short time window, the bursty arrival at the P4-switch can lead to higher peak rates to be handled. For example, if status

messages from all the robots arrive within a time window of 1ms (50% of the 2ms sending interval), the observed temporal rate could be 3 Gbps or higher. One can observe that these arrival rates can easily be served by currently available P4-hardware including both smartNICs, DSCs, and P4-switches. 2) **The robot-control traffic can be separated** from other traffic either by assigning dedicated ports and/or pipes to robot traffic or using simple priority queues giving higher priority to industrial traffic than background packets. Note that priority queueing is supported by most of the networking elements (also including non-P4-programmable ones). This scenario is examined in more detail in Appx. C.

We also tested our pipeline enforcing the packet resubmission at ingress, but it had no visible effect on the latency distribution. Finally, we repeated all the delay measurements with generating robot traffic at 100 Gbps from Server-B, but it has no effect on the observed latency at Server-A.

7.2 Synchronization measurements

Robot to robot collaboration is an important use case in any industrial robot cell deployment. To speed up the assembly process a usual deployment contains a robot arm moving the part to be worked with into various reachable positions for the other arm that has various grippers and executes a specific assembling order. The two arms need perfect synchronization otherwise the resulting product is faulty.

In this operational experiment, we launch the real UR5e robot arm and an instance of the URSim robot emulator, both are controlled by the switch and we start the trajectories in sync and out of sync. Fig. 9 shows the time shift between the start times of the two robots. The experiment was repeated 20 times. In the synchronized case, we created a single entity from the two robot arms with 2×6 joints and launched the trajectory by adding an entry to the TrajectorySwitcher table. The result is a fully synchronized operation as depicted by blue in the figure. In the non-synchronized case, the two entries are inserted independently to start the two robots. Note that we observe an 8 ms time shift in the worst case that is comparable with the control frequency of the robots (125 Hz) and can simply be caused by the 8 ms real-time window of the robots. Though the observed time-shift is basically negligible for two robot arms, we assume that it may be much more significant if a larger number of robots is launched independently.

7.3 Accuracy at stop position

The accuracy and repeatability of a robotic arm are essential key performance indicators (KPIs) that need to be maintained even in a cyber-physical-system, i.e., remote control over the network. The basic pick and place, and palletizing use cases mostly depend on them. It is a bare minimum that the proposed system works well in these use cases.

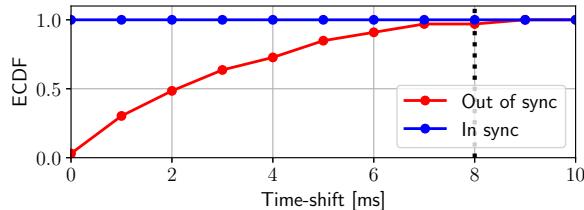


Figure 9: Time shift.

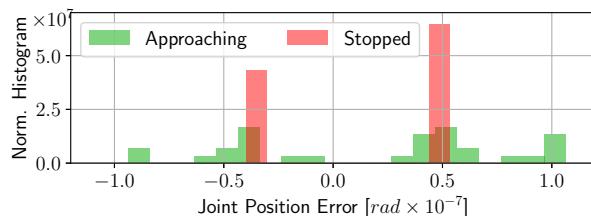


Figure 10: Joint position error at stop.

We performed experiments to analyze the accuracy of the control at the stop position (at the end of the trajectory) of the robot arm. One can observe in Fig. 10 that the error from the expected joint position was about $0.5 \times 10^{-7} \text{ rad}$ which was caused by the applied number representation (see Sec. 5.3). The green bars illustrate the deviance when the joint speed is not absolute zero, but the joint is close to its target position. Note that $0.5 \times 10^{-7} \text{ rad}$ error in the joint position corresponds to $0.5 \mu\text{m}$ with a 1m long robot arm. In a robot arm with multiple joints, the cumulative position error is still in the order of micrometers.

7.4 Accuracy along the trajectory

The assembling quality and the endurance of a product mostly depend on the quality of gluing, welding and painting work. To ensure this, the robot needs to be accurate not only in the goal positions but all along the planned trajectory.

Though the proposed implementation is highly configurable and supports the fine tuning of the applied P-controller, more advanced controllers (e.g., PID) can obviously provide more precise control. In this experiment, we measure the accuracy of the robot head at the trajectory points and compare the results to the PID-based velocity control of ROS. Both ROS and the emulated robot run on the same server, representing ideal circumstances for ROS-based control.

Fig. 11a and 11b depict the TPs (green points) as well as the path of the tool at the end of the robot arm (solid curves) for controls based on ROS and P4-Switch, resp. Both paths show a similar character. The accuracy of the two solutions is presented in Fig. 11c. ROS's fine-tuned PID-controller provides a 0.1 mm accuracy in the worst case which is 3.2

mm in the case of our proof-of-concept P-controller. The median accuracy values are 0.04 mm and 2.23 mm for ROS and P4-Switch, respectively.

7.5 Continuous table management

Industry 4.0 introduces the concept of agile robot cell control that requires fast reaction to external events, e.g., based on camera or force sensor feedback. Ensuring safety during robot and human collaboration is also critical. It is essential for the proposed system to react fast to external triggers.

In this experiment, we used the same measurement setup as in Sec. 7.1. We generated robot traffic at 10 Gbps and sampled the latency every 1 ms. In the beginning, we loaded 3.4K trajectory points to the switch and then started the operation. In every 1 second, we add 1.6K new TPs and remove 10K outdated TPs, illustrating the case when the switch is only used as a playback buffer, and the trajectory segments are loaded incrementally, while the old points are removed. Note that inserting a trajectory point with 6 joints requires the insertion of 12 entries into two exact-match tables. According to realistic scenarios, a trajectory normally contains 5-10 points in a second. Fig. 12 illustrates the latency samples and their moving average (on the bottom), and also shows the number of trajectory points (black) loaded into the switch in time, marking the insertion (blue) and removal (red) phases (on the top). One can observe that the insertion does not affect the packet processing latency in this scenario.

8 Discussion on Possible Deployment

Apart from the theoretic aspect and the successful proof study that the proposed system is feasible to deploy, the possibility of a real industrial deployment is much dependent on the cost factors. A simple calculation reveals that a Tofino-based router costs approx. 9500 USD and it can serve up to 500-1000 robots in parallel which means that the cost of controlling a robot is less than 10-20 USD. It is less than applying mini PCs for hobby use, e.g., Raspberry Pis, and far less than certified industrial robot controllers or routers. The energy consumption of the proposed setup is expected to be much lower than the sum of industrial routers and robot controllers. A network device has better transported traffic per watt ratio than a general purpose computation device that the current robot controllers contain. Though in industry, usually low performance, but reliable old CPUs are applied. According to [3] a programmable switch will result in about 14% extra cost, compared to a non-programmable one, due to the larger area requirement for transistors. It is an interesting aspect if energy saving can be achieved by switching non-working elements on and off. One can observe that the capabilities of a Tofino are far more than what is required for the robot control use case. The utilization of the device can be improved by only

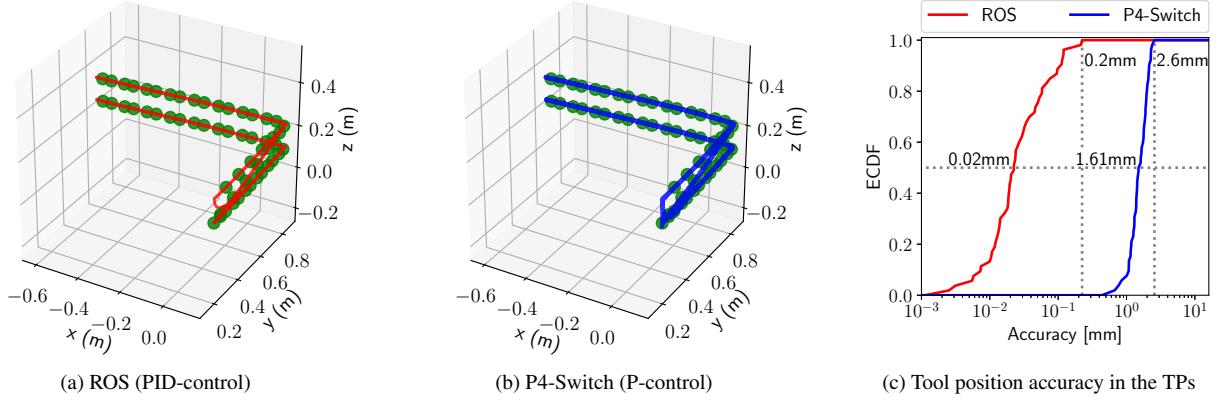


Figure 11: Path of the robot arm tool in the Cartesian-space and the observed accuracy in the TPs.

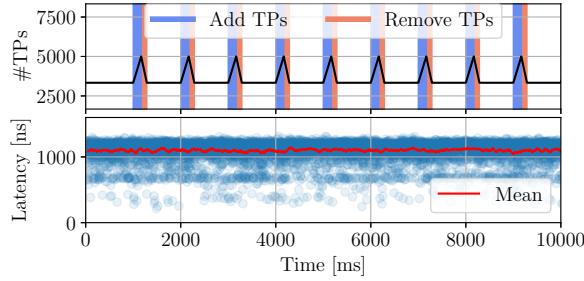


Figure 12: Dynamic insertion and removal of trajectory points.

dedicating a part, e.g., a quarter of the switch to robot control while other parts can work on other tasks (e.g., routing).

The current workflow of a robot is that when it is switched on, it starts streaming out the internal status messages at a constant rate. Production cells are expected to operate 24 hours a day, so little can be done dynamically, apart from the fact that the default power consumption is significantly lower than current systems. The typical power consumption of a Tofino switch is around 110 W [16], while an average server requires 400-600 W. A modern server CPU alone can consume 165 W (Intel Xeon Gold 6348H Processor) at full load. If we compare the costs of purchasing a server with similar processing power and memory to the cheapest P4-Switch, the difference is not too significant. For a brief discussion on x86 alternatives see Appx. B. Also note that this is the first commercially available version of the Tofino switch, and as more and more new models appear and become more available, prices are expected to drop.

Considering the edge-cloud deployment scenario mentioned in Sec. 1, offloading computations that are simple but have real-time requirements that cannot be satisfied in a virtualized environment also have practical benefits. In this case,

distributed service cards or smart NICs with P4 programmability could be more cost effective solutions than a Tofino-based switch. They cost around 1500-3000 USD, also enables line rate (10-40 Gbps) processing with sub-millisecond response time, and have a typical power consumption of 20-50 W.

9 Conclusion

In this paper, we have introduced the first in-network control system that uses P4-programmable network devices for not just triggering events based on threshold values, but to do low-level real-time velocity control for highly delay-sensitive robotic arms that can be used in industrial automation. With several experiments, we have proved that our system satisfies the most crucial factors of industrial robot control. We measured the latency and observed that it meets the requirements needed for real-time control even during the constant insertion and deletion of lookup table entries. We witnessed a maximum of an 8 ms time shift in the worst-case scenario between synchronous robots, making them fully capable of collaboration. We evaluated the end-position precision per joint to be under $0.5\mu\text{m}$ for a 1 m long robot arm, while the accuracy along the whole trajectory to be lower than 2.6 mm in the worst-case.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback on earlier versions of this paper. S. Laki and P. Vörös also thank the support of the "Application Domain Specific Highly Reliable IT Solutions" project that has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme.

References

- [1] Mehdi Bennis, Mérouane Debbah, and H. Vincent Poor. Ultra-Reliable and Low-Latency Wireless Communication: Tail, Risk and Scale. *Corr*, abs/1801.01270, 2018.
- [2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.
- [3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, August 2013.
- [4] Fabricio E Rodriguez Cesen, Levente Csikor, Carlos Recalde, Christian Esteve Rothenberg, and Gergely Pongrácz. Towards low latency industrial robot control in programmable data planes. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 165–169. IEEE, 2020.
- [5] David Coleman, Ioan Alexandru Sucan, Sachin Chitta, and Nikolaus Correll. Reducing the barrier to entry of complex robotic software: a moveit! case study. *ArXiv*, abs/1404.3785, 2014.
- [6] Comau 5G deployment. <https://www.ericsson.com/en/reports-and-papers/ericsson-technology-review/articles/industrial-automation-enabled-by-robotics-machine-intelligence-and-5g>, 2017.
- [7] P. Danielis, J. Skodzik, V. Altmann, E. B. Schweissguth, F. Golatowski, D. Timmermann, and J. Schacht. Survey on real-time communication via ethernet in industrial automation environments. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8, 2014.
- [8] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference, IMC ’15*, page 275–287, New York, NY, USA, 2015. Association for Computing Machinery.
- [9] Y. Guo, X. Hu, B. Hu, J. Cheng, M. Zhou, and R. Y. K. Kwok. Mobile cyber physical systems: Current challenges and future networking applications. *IEEE Access*, 6:12360–12368, 2018.
- [10] ISO: International Organization for Standardization. 1998. Manipulating industrial robots – Performance criteria and related test methods, NF EN ISO9283. <https://www.iso.org/standard/22244.html>, 1998.
- [11] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg. A survey of research on cloud robotics and automation. *IEEE Transactions on Automation Science and Engineering*, 12(2):398–409, April 2015.
- [12] Thomas Kohler, Ruben Mayer, Frank Dürr, Marius Maaß, Sukanya Bhowmik, and Kurt Rothermel. P4cep: Towards in-network complex event processing. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 33–38, 2018.
- [13] D. W. McKee, S. J. Clement, J. Almutairi, and J. Xu. Massive-scale automation in cyber-physical systems: Vision amp; challenges. In *2017 IEEE 13th International Symposium on Autonomous Decentralized System (ISADS)*, pages 5–11, March 2017.
- [14] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.
- [15] Jan Rüth, René Glebke, Klaus Wehrle, Vedad Causevic, and Sandra Hirche. Towards in-network industrial feedback control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 14–19, 2018.
- [16] Nik Sultana, John Sonchack, Hans Giesen, Isaac Peidisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.
- [17] Universal Robot 5e. <https://www.universal-robots.com/products/ur5-robot/>, 2020.
- [18] URCap. <https://www.universal-robots.com/about-universal-robots/news-centre/launch-of-urcaps-the-new-platform-for-ur-accessories-and-peripherals/>, 2014.
- [19] URScript. <https://www.universal-robots.com/how-tos-and-faqs/how-to/ur-how-tos/ethernet-socket-communication-via-urscript-15678/>, 2017.
- [20] Jonathan Vestin, Andreas Kessler, and Johan Åkerberg. Fastreact: In-network control and caching for industrial control networks using programmable data planes. In

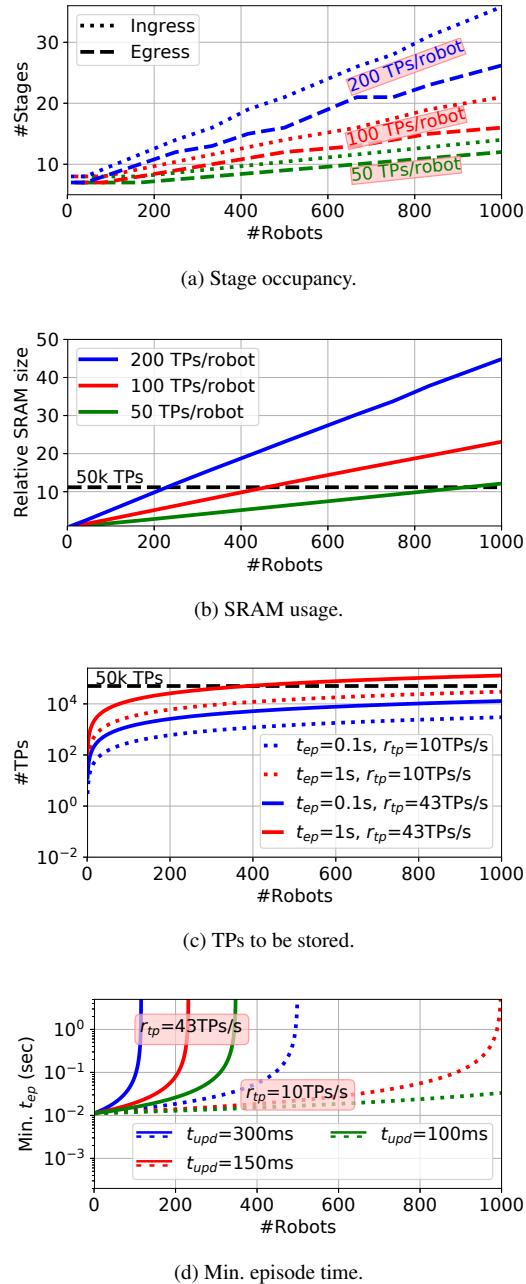


Figure 13: Resource usage of different setups with variable number of robot arms, different trajectory granularity and control plane speed.

A Scalability estimation

We have carried out various micro-benchmark measurements to estimate the scalability of the proposed in-network robot control method in terms of both computational, memory resources and the speed of interaction between data and control planes. Fig. 13a-13b show the first experiment group where the number of trajectory points stored by the switch for each robot is varied. We consider three settings: 50, 100 and 200 TPs/robots. Note that in an average case, a trajectory consists of 10 TPs in every second. These numbers can be interpreted in two ways: 1) this is the number of TPs in the entire trajectory of a given robot, 2) the TPs related to a trajectory episode as discussed in Sec. 7.5 (Note that an episode of n TPs requires space for storing at least $3n$ TPs in the pipeline: expired episode to be deleted, active episode that is under execution, upcoming episode that will be executed after the active one). The TPs are stored in exact tables of the pipeline that are mapped to the SRAM. One can observe that both the number of stages and the SRAM usage scale linearly with the number of robot arms. The increase in SRAM usage expresses the rising number of table entries (6 entries in two tables for each TP). Note that the SRAM usage could be the same or similar in other P4-targets (e.g., smartNICs, DSCs). However, the increase in the number of stages is directly related to the physical structure of underlying P4-device, and could vary from target to target. In our case, SRAM is distributed among stages, and if the table is too large, it is spread among multiple stages, increasing the stage occupancy. Note that the P4-switch we used for evaluation is able to store at most 50K TPs without any limitation. The TCAM usage of the proposed method is limited and predictable. Tables used for approximating the calculations in the PID-like controller are mapped to the TCAM area whose size only depends on the required control precision (Sec. 5.3).

Fig. 13c-13d focus on the dynamic use case discussed in Sec. 7.5, showing the relationship between resource usage (#TPs), the number of robots, the length of episodes (t_{ep}), the granularity of trajectories (r_{tp} : normal usage with 10 TPs/s; fine-grained movement with 43 TPs/s) and the time (t_{upd}) needed for the control plane for updating tables storing TPs in data plane. Note that t_{upd} in the figure illustrates the time needed for adding and removing 1.6K TPs (2x10K entries), and according to our measurements the update time scales linearly with the number of TPs, but it cannot go below 1 ms. One can observe that in this dynamic scenario the speed of the control plane determines both the minimum length of a trajectory episode and the maximum number of robots to be controlled for a given r_{tp} . In our prototype control plane t_{upd} is almost 300 ms, and thus for $r_{tp}=10$ TPs/s 500 robot arms can be controlled with $t_{ep} \geq 1$ s. One can also see that it requires less memory resources than the 50K limit and thus the speed of the control plane has become the bottleneck in this case, limiting the number of robots to be integrated.

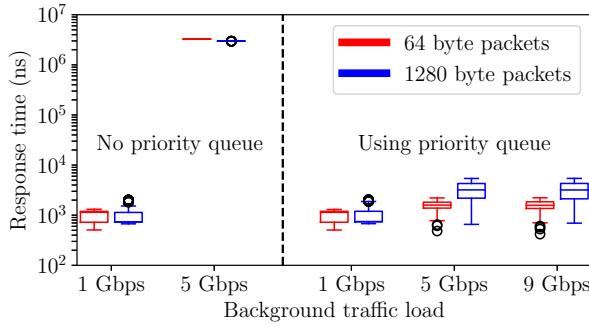


Figure 14: The observed response times in case of a 5Gbps bottleneck

Though the first generation P4-programmable hardware targets including smartNICs and switches have well-known limitations, they can still be used for offloading real-time computational tasks. We think that the next generation of such devices that are on the horizon will give a momentum to the use of in-network computing and enable supporting various applications that require ultra-low latency, high-throughput real-time and/or predictable performance. We believe that real-time cloud and edge cloud applications like robot control are one of the potential use cases that can benefit from in-network computation.

B Comparison to x86

We have shown in the previous section that a single P4-switch can be scaled up to control 500 or even 1000 robot arms, depending on the use cases and settings. However, the low-level velocity vector calculation can also be separated from the industrial controller and offloaded to a dedicated computer in a traditional scenario. In this section, we consider a distributed ROS deployment where the low-level control is coordinated by a robot-driver node in ROS. For each robot arm, a dedicated process is executed to receive status messages, perform the calculations and send velocity commands. Robot-driver nodes require real-time Linux kernel to ensure the timing requirements. We evaluated the driver node of UR5e in a multi-core server equipped with two CPUs (2x Intel Xeon CPU E5-2630 2.30GHz 6C/12T, 32GB RAM). A single control process resulted in 0.19 CPU usage (out of 12, the number of logical cores) in idle state which went up to 0.42 after the robot arm was connected. The CPU usage scaled linearly with the number of robot arms. The CPU limit was reached with 45 emulated robot arms after that ROS processes started interfering each other. The total system load was around 95%.

C Interference with regular network traffic

In Sec. 7.1, we have shown that the response time in under-loaded situations is around $1-2\mu\text{s}$. Though we think that the separation of control and regular traffic could be possible in most environments, in this section we investigate how regular traffic with different load level affects the processing of control messages. To make the interference more visible, the port rates of the switch are limited to 1Gbps, 5Gbps and 9Gbps. 90% of the test traffic is regular traffic (i.e., non robot control packets) while the remaining 10% consists of robot control messages. The load level is varied from 1Gbps to 9Gbps. The same testbed is used as in Sec. 7.1.

In Fig. 14, we have created an artificial bottleneck of 5Gbps by rate limiting the used egress port. The left side of the figure depicts the case when the regular and robot control traffic is not separated from each other. The packet size in the regular traffic is either 64 or 1280 bytes, marked with red or blue, resp. One can observe that when the arrival rate is 1 Gbps which is much smaller than the bottleneck capacity, the response time is around $1\mu\text{s}$ as in our previous analysis. Note that no packet loss is experienced in this case. However, when the arrival rate of the test traffic is increased to 5Gbps, the outgoing port starts being congested, packets accumulate in the buffer and thus the observed latency of robot control packets significantly increases ($3 \times 10^6 \text{ ns} = 3\text{ms}$) due to queueing and a part of the packets is lost. One can note that the increased response times and packet losses degrade the performance of our robot control method, making it unreliable. With an arrival rate of 9Gbps, almost all robot control packets are lost due to congestion. Regular traffic with high intensity has a clear impact on the robot control traffic if they share the same buffer.

However, most P4 programmable devices allow to define multiple queues for each egress port and apply strict priority scheduling between them. As depicted on the right side of Fig. 14, directing regular and robot control traffic into two separate buffers, applying strict priority scheduling between them and giving higher priority to robot control traffic can easily solve the problem of interference. Even in extreme congestion situations (5Gbps or 9Gbps background load) the response times still remain in sub-millisecond order with zero packet loss.

Note that we have obtained similar results for other bottleneck capacities.

Enabling IoT Self-Localization Using Ambient 5G Signals

Suraj Jog[†], Junfeng Guan[†], Sohrab Madani[†], Ruochen Lu^{*}, Songbin Gong[†], Deepak Vasisht[†], Haitham Hassanieh[†]

University of Illinois at Urbana Champaign[†], University of Texas at Austin^{}*

Abstract – This paper presents *ISLA*, a system that enables low power IoT nodes to self-localize using ambient 5G signals without any coordination with the base stations. *ISLA* operates by simply overhearing transmitted 5G packets and leverages the large bandwidth used in 5G to compute high-resolution time of flight of the signals. Capturing large 5G bandwidth consumes a lot of power. To address this, *ISLA* leverages recent advances in MEMS acoustic resonators to design a RF filter that can stretch the effective localization bandwidth to 100 MHz while using 6.25 MHz receivers, improving ranging resolution by 16×. We implement and evaluate *ISLA* in three large outdoors testbeds and show high localization accuracy that is comparable with having the full 100 MHz bandwidth.

1 Introduction

Recent years have witnessed a tremendous growth in the number of connected IoT devices, with surveys projecting up to 31 billion deployed IoT nodes by 2030 [38]. With such ubiquitous deployment of IoT nodes, the ability to localize and track these nodes with high accuracy is essential for many applications. For example, in data driven agriculture, it can enable real time micro-climate monitoring and livestock tracking [39]. In smart cities, IoT sensors are deployed throughout the city for tasks such as air quality monitoring, tracking buses, trains, and cars, and monitoring the structural health of infrastructure [22]. In the era of Industry 4.0, it can also enable wide area inventory tracking and facilitate factory automation [24].

Today, the most prevalent outdoors localization technology is GPS which is mainly used in cars and mobile phones. However, off-the-self GPS chips can consume about the same power as the entire IoT device, thus reducing the battery life to half in addition to the extra hardware costs [5]. Due to this, past work has proposed the use of cellular networks or dedicated IoT base stations for localization [9, 27]. These solutions, however, either achieve very low resolution of 100s of meters [9, 18] or require active participation of the base stations to jointly compute the location or tightly synchronize the base stations [27, 40, 45]. Realizing such solutions in practice requires the cooperation of cellular providers to bear the additional cost of modifying the base stations and a back end server to support the localization feature.

In this paper, we ask whether an IoT device can accurately localize itself simply by listening to ambient 5G cellular signals, without any coordination with the 5G base stations? Doing so would allow us to easily deploy self-localizing IoT nodes in wide areas without the need to modify the cellular base stations or deploy new base stations for localization.

5G cellular networks present unique opportunities for enabling accurate localization. First, the small cell architecture in 5G networks will lead to a very high density of 5G base stations, with up to 40 to 50 base stations deployed per square km [15], thereby allowing us to leverage more anchor points in the network for increased localization accuracy. Second, the 5G standard is designed to support very high data rates and can have OFDM signals spanning up to 100 MHz in bandwidth in the sub-6 GHz frequency range, and up to 400 MHz bandwidth in the mmWave frequency range [37]. Such large bandwidth can be used for accurate localization. To see how, consider the 5G OFDM signal shown in Fig. 1(a) where data bits are encoded in N frequency subcarriers. We can use the preamble which contains known bits to compute the channel impulse response (CIR) by taking an inverse FFT. The CIR in Fig. 1(a) shows the Time-of-Flight (ToF) of different signal paths. Estimating the ToF from few base stations allows us to localize the device. The larger the bandwidth of the signal, the higher the resolution. In fact, we can achieve a resolution of 3 meters for 100 MHz and 0.75 meters for 400 MHz signals.¹

Leveraging these opportunities, however, is challenging since power-constrained and low-cost IoT nodes cannot capture the large bandwidth of the 5G signals. They are equipped with low-power and low-speed Analog-to-Digital Converters (ADCs) that can only capture a narrow bandwidth. In fact, while IoT has been one of the cornerstone applications in the design of 5G, it is only supported in narrowband chunks for low data rate applications [2, 3]. Therefore, while the 5G standard does allocate higher bandwidth (up to 400 MHz) for mobile broadband and high data rate applications, IoT nodes can capture only a very small fraction of this bandwidth ($\sim 20 \times$ smaller [37]). As a result, they significantly lose out on the ToF resolution that was made possible by the high bandwidth 5G signals as shown in Fig. 1(b). Moreover, it is infeasible to measure the absolute time-of-flight without any coordination or synchronization with the base stations.

In this paper, we present *ISLA*, a system that enables IoT Self-Localization using Ambient 5G signals. *ISLA* does not require any coordination with or modifications to the base stations. The key enabler of *ISLA* is the use of MEMS (micro-electro-mechanical-system) acoustic resonators. Past work [11, 12] has demonstrated that we can use such MEMS resonators to design new kinds of RF filters that look like a spike-train in the frequency domain, as shown in Fig. 1(c). To understand how we can leverage such MEMS spike-train filters, consider the 5G OFDM signal shown in Fig. 1(a).

¹The resolution is computed as c/B where c is the speed of light and B is the bandwidth of the signal.

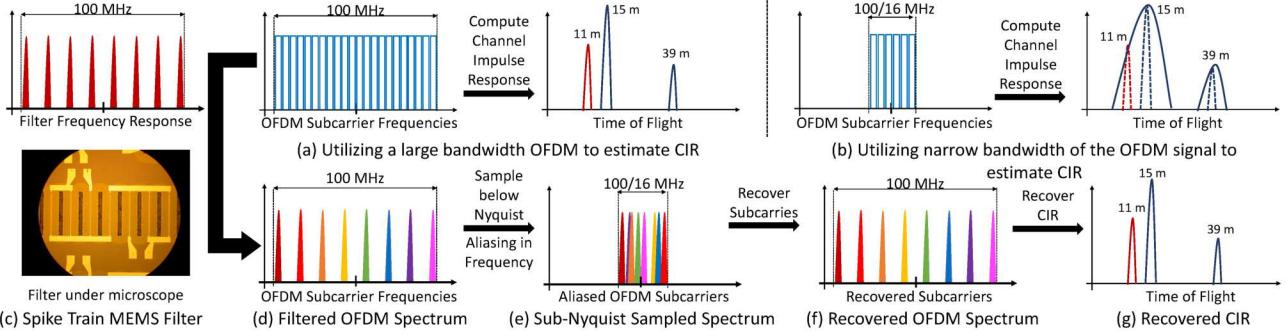


Figure 1: ISLA’s pipeline. (a) wideband OFDM signal and its corresponding CIR. (b) narrowband OFDM signal and its corresponding lower resolution CIR. (c) ISLA’s spike train MEMS filter that sparsifies the wideband signal. (d-f) follow the signal journey through ISLA’s pipeline that recovers the original CIR.

Passing this signal through the filter allows us to keep a few subcarriers of the wideband OFDM symbol while suppressing all other subcarriers as shown in Fig. 1(d). There are two important features of the resulting signal: (1) Since the remaining subcarriers that are passed by the filter span the entire wideband, we should, in principle, be able to recover the channel impulse response at the same high resolution of the original signal. (2) Since the remaining subcarriers create a sparse signal in the frequency domain, it should be possible to recover these subcarriers by sampling the signal below the Nyquist sampling rate using the same low-power low-speed ADCs on the IoT nodes.²

However, recovering the channel impulse response from a signal sampled with the low-speed ADCs is non-trivial. First, sampling the signal below the Nyquist rate leads to aliasing in the frequency domain as shown in Fig. 1(e). Some subcarriers might collide by aliasing on top of each other making it hard to recover these subcarriers. Past work in sparse recovery addresses this problem by using two co-prime subsampling rates [16]. Unfortunately, we do not have the flexibility to choose co-prime subsampling factors. In fact, since the number of OFDM subcarriers in the 5G standard is a power of 2 (e.g. 1024, 2048, 4096), we can only subsample the signal by powers of 2 otherwise the values of the subcarriers will be corrupted as we prove in section 5.³ To address this, we carefully co-design the MEMS hardware with the recovery algorithm. In particular, we jointly optimize the filter shape (spacing between peaks, width of each peak, frequency span) with the subsampling rate to minimize the number of colliding OFDM subcarriers as we describe in detail in section 5.

Second, the recovered OFDM subcarriers are not uniformly distributed across the wideband bandwidth. This is because non-idealities in the MEMS filter make it hard to design a uniform spike train like the one shown in Fig. 1(c). As a result, we can no longer recover the CIR using standard super-resolution algorithms like MUSIC with spatial smoothing [21, 44] as they require uniform measurements. Instead, we formulate an inverse optimization problem that accounts for non-idealities

and optimizes the CIR in the continuous time domain to achieve super resolution as described in Sec. 5.

Finally, while the above can provide very precise ToF measurements, these ToF estimates are not going to capture the true time taken by the signal to travel between the base station and the IoT device. This is because the 5G base stations are not time-synchronized with each other or the IoT device. To localize the device without any synchronization with the base station, ISLA leverages a second antenna on the receiver to compute the differential ToF of the propagation paths. While the absolute ToF measurements are corrupted by synchronization offsets, these offsets are constant across the 2 antennas on the IoT node, and hence can be eliminated by subtracting the measurements from the 2 antennas. Using this differential ToF at the IoT receiver, we show in section 7 that with measurements from four or more base stations, the IoT device can localize itself regardless of its orientation. We integrate our approach into a full system that addresses additional system challenges such as figuring the base station ID and accounting for carrier frequency offsets.

Evaluation: We implemented and evaluated ISLA indoors for microbenchmarks and outdoors for overall localization performance. We ran experiments in three outdoor settings:(1) Between campus buildings (52 m×85 m), (2) a large parking lot (240 m×400 m), and (3) an agricultural farm (480 m×860 m). We use USRP X310 radios as base stations that can transmit high-bandwidth packets of 100 MHz. Our custom IoT nodes are equipped with 2 antennas and subsample the 5G signals at 6.25 MS/s which is 16× below the Nyquist rate. We fabricated a MEMS spike-train filter operating at a center frequency of 400 MHz and used it to demonstrate accurate reconstruction of the channel impulse response. However, due to significant interference at the 400 MHz band outdoors in our city, we ran experiments at 1 GHz and applied the filter response in digital. Our results reveal that with 5 base stations in range, ISLA can achieve a median accuracy of 1.58 m on campus, 17.6 m in the parking lot, and 37.8 m in the farm where the IoT node can be as much as 500 meters away from most base stations. For the parking lot testbed, the accuracy improves to 9.27 m with 15 base stations and 4.26 m with 25 base stations in range. We compare ISLA’s localization

²Note that the MEMS filter is passive and does not consume any power.

³For example, for a 100 MHz OFDM signal, we can only sample at 50 MS/s (2×), 25 MS/s (4×), 12.5 MS/s (8×), 6.25 MS/s (16×), ...

approach with several baselines [9, 21, 43] and show up to $4\text{--}11\times$ higher localization accuracy. Finally, we show that *ISLA* achieves a comparable performance to having a full 100 MHz receiver while using a $16\times$ lower sampling rate.

Contributions: We make the following contributions:

- We present, to the best of our knowledge, the first system that allows IoT nodes to localize themselves using ambient 5G signals without any coordination with the base stations.
- We demonstrate the ability to reduce the sampling rate by $16\times$ while retaining the benefits of high bandwidth 5G signals by leveraging recent advances in MEMS RF filters.
- We implement and evaluate *ISLA* to demonstrate accurate localization in 3 outdoor settings.

2 Related Work

Localization has been extensively studied in cellular, WiFi, and IoT networks. Our work differs from past research in that it is the first to enable self-localization using ambient 5G signals without requiring coordination with the base stations.

A. Cellular Based Localization: Several studies [9, 17, 18, 29, 33] have proposed to use nearby cell tower information and statistics in order to localize a mobile device. These methods, however, have a median accuracy of around 100 to 500 meters, and are mostly useful for very coarse localization. To improve localization accuracy, [4, 35] propose to combine WiFi APs with cellular base stations. Despite their relatively higher accuracy, these methods require fingerprinting the surroundings and as such require extensive training and do not generalize to new locations. More recent work exploits massive MIMO and millimeter wave for localization in 5G [30, 31, 42]. However, all of this work requires coordination with base stations and assumes the devices can capture the entire bandwidth of the 5G signals which does not work for IoT devices.

B. IoT Based Localization: [5] leverages TV whitespaces to achieve high localization accuracy for LoRa IoT devices. However, it requires all base stations to be tightly synchronized at the physical layer (time and phase) in order to measure TDoA (Time Difference of Arrival). Recent work [27] designs low power backscatter devices that leverage LoRa for localization to achieve high accuracy. However, the system mainly targets indoor applications where software radios can be deployed as base stations to sample the I/Q of the signal and localize the IoT node. Moreover, its current system design [27] supports only a single node. The authors of [34] propose an outdoors localization technique for SigFox IoT devices based on fingerprinting. However, as mentioned earlier, fingerprinting requires constant training and cannot scale to new environments. Finally, there is a lot of work on using UWB or RFID nodes for localization [10, 13, 41]. However, these works focus on indoors and short range as the range of UWB and RFIDs is limited to 10-30 meters [7, 14].

C. IoT Self-Localization: LivingIoT [19] enables self-

localization on IoT nodes. It designs a miniaturized device that can be carried by a bumblebee and uses backscatter for communication. The node localizes itself by extracting the angle to the Access Point from the amplitude measurements using an envelop detector. The technique, however, requires the APs to switch the phase across two antennas to change the received amplitude at the IoT node, and hence, cannot be applied to 5G without modifying the base stations. [26] enables self-localization by placing a camera on a WISP RFID but only operates within a range of 3.6 m from the RFID reader.

D. WiFi Based Localization: There has been a lot of work on indoor localization using WiFi [6, 21, 25, 32, 40, 43, 44, 46, 47]. The closest to our work are [21, 40, 43] which estimate the channel impulse response (CIR) and time of flight (ToF) from the WiFi access point (AP). Chronos [40] hops between WiFi channels to compute the CIR at high resolution. However, it requires tight timing coordination with the AP to compensate for carrier frequency offset (CFO) and ensure phase coherence across the measurements. *ISLA*, on the other hand, captures measurements from many frequencies across a wideband without hopping by using the MEMS filter, and hence, does not require any coordination with the base stations. SpotFi [21] combines measurements across antennas with large WiFi bandwidth to separate Line of Sight (LoS) path from multipath reflections in the CIR using MUSIC along two dimensions: ToF and Angle of Arrival (AoA). mD-Track [43] also incorporates Doppler shifts and Angle of Departure (AoD) in addition to ToF and AoA and iteratively refines the CIR to achieve a better estimate of the LoS path. In section 10, we adapt SpotFi’s and mD-Track’s CIR estimation algorithms to our setting and demonstrate that *ISLA*’s algorithm achieves $4\text{--}11\times$ higher accuracy. It is worth noting, however, that for our application, these past works cannot benefit from the doppler or AoA/AoD dimensions.

E. MEMS Filter: Recent work has used MEMS spike-train filters for the application of wideband spectrum sensing [12]. However, [12] can only detect signal power at different frequencies and cannot recover complex I and Q samples needed for estimating the CIR. Furthermore, [12] deals with collisions resulting from aliasing by using co-prime sub-sampling rates. Such approach does not apply in the context of 5G OFDM signals, since, as we show in section 5 the sub-sampling factor can only be a power of 2. *ISLA* instead co-designs the hardware filter together with sampling rate to avoid collisions.

3 Background

A. Spike-Train MEMS Filters: Our work builds on recent advances in MEMS RF filters. MEMS filters can work between a few MHz and 30 GHz and can be integrated with ICs to form a chip-scale RF front-end solution for IoT devices. Past work on MEMS RF filters optimize for filters with a single passband [36, 48], however, the MEMS filter used by

ISLA leverages MEMS resonators that have an assortment of equally spaced resonance frequencies to create a spike train in the frequency domain as shown in Fig. 1(c).

A MEMS filter works by leveraging the inverse piezoelectric effect to convert RF signals into acoustic vibrations for filtering and processing. It then converts acoustic waves in the device back to the RF signals through piezoelectric effect. In this process, the frequency filtering is achieved because not all frequencies can be efficiently converted between RF and acoustic domains. Frequencies that match the resonance frequencies of the piezoelectric structure can go through the conversions with little loss, while other frequencies are filtered out. Hence, the spike train frequencies can be designed by changing the dimension of the piezoelectric material in the MEMS device as well as the placement of electrodes shown under the microscope in Fig. 1(c).

B. Wireless Channel Impulse Response (CIR): The wireless channel can be modeled as the superposition of the signal along all the different paths it takes to travel from the transmitter to the receiver. The channel at frequency f_i can be written as: $h_i = \sum_{l=1}^L a_l \exp^{-j2\pi f_i d_l / c}$, where L is the number of propagation paths between the transceivers, d_l is the distance traversed by path l , a_l is the complex path attenuation of path l , and c is the speed of light.

In OFDM systems, data is transmitted over multiple frequency subcarriers $\{f_0, \dots, f_{N-1}\}$. If the frequency spacing between these subcarriers is Δf , then the bandwidth spanned by the signal is $B = \Delta f \times (N - 1)$. Now, given the channel measurements $\{h_0, \dots, h_{N-1}\}$ across these frequencies, the Channel Impulse Response (CIR) can be computed as the inverse FFT of the channel measurements.

$$CIR(\tau) = \sum_{n=0}^{N-1} \left(\sum_{l=1}^L a_l \exp^{-j2\pi \frac{d_l}{c} f_n} \right) \exp^{j2\pi \tau f_n} \quad (1)$$

where $\tau = \{\frac{0}{B}, \dots, \frac{(N-1)}{B}\}$ seconds. There are two important things to note here. First, the resolution in Time-of-Flight in the CIR is $1/B$ seconds, that is inversely proportional to the bandwidth B . Hence, larger bandwidth results in higher ToF resolution and more accurate ranging. Second, the maximum unambiguous ToF that can be measured from the CIR is $\frac{(N-1)}{B} = 1/\Delta f$ seconds. This means, if some physical propagation path in the environment has ToF $> 1/\Delta f$ then it would alias and appear at a different tap value in the estimated CIR in Eq. 1. For 5G OFDM signal with $B = 100$ MHz bandwidth and $\Delta f = 60$ kHz, we have a resolution of 10 ns (3 meters) and a range of 16.6 μ s (5 km).

4 System Overview

ISLA enables self-localization on narrowband IoT devices by leveraging the MEMS spike-train filter to capture ambient wideband 5G signals. *ISLA* consists of 3 main components:

(1) Capturing the wideband 5G OFDM signal using the MEMS filter: The received 5G signal is passed through the

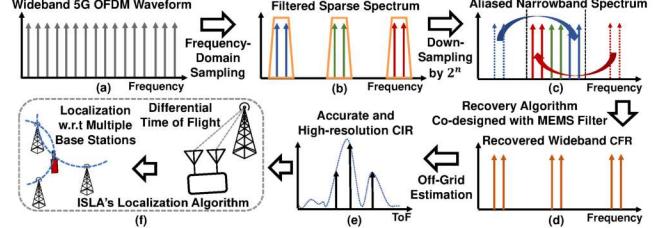


Figure 2: Overview showing the flow of *ISLA*'s system

MEMS filter which samples the OFDM symbol in the frequency domain. Specifically, the MEMS filter passes the OFDM frequency bins that align with the filter passbands while suppressing all other frequency bins. The resulting output from the filter is a sparse spectrum as shown in Fig. 2(b). This sparse signal is then subsampled by the narrowband IoT device significantly below the Nyquist rate ($16\times$ lower) which results in aliasing the remaining subcarriers into the narrowband as shown in Fig. 2(c). We co-design the filter hardware with the recovery algorithm to easily reconstruct the wideband OFDM subcarriers as we describe in section 5.

(2) Super-Resolution CIR Estimation: Using the recovered wideband channel measurements, *ISLA* then reconstructs a high resolution Channel Impulse Response (CIR) by leveraging its super-resolution algorithm which estimates the off-grid positions of the propagation paths as described in Section 6. This high-resolution CIR allows *ISLA* to filter out the LoS path from the multipath in the channel for high resolution time-of-flight estimation as shown in Fig. 2(e).

(3) Localization Algorithm: Since the IoT node is not synchronized with the base station, the measured ToF will be corrupted by a timing offset. To address this, *ISLA* leverages two antennas on the IoT device and computes the differential CIR across the antennas to eliminate the synchronization offsets. This results in the locus of the IoT device to lie on a circle that is defined by the locations of the base stations and the angle subtended by the base stations at the IoT device's location, as we explain in Section 7. Thus, by looking at the intersection of such circles, we can accurately infer the position of the IoT device as shown in Fig. 2(f). Finally, we show how to integrate *ISLA* with the 5G-NR standard by addressing additional system challenges in section 8.

5 Capturing 5G Signals Using MEMS Filter

ISLA leverages the MEMS spike-train filters to capture the wideband channel measurements on a narrowband receiver. We explain this sensing process through Fig. 2. Consider a preamble OFDM symbol transmitted from the base station with N subcarrier frequencies at $\{f_0, \dots, f_{N-1}\}$, shown in Fig. 2(a). Let the received time domain symbol be $x(t)$ and its frequency domain representation be $X(f)$. We have $X(f) = \sum_{n=0}^{N-1} c_n h_n \delta(f - f_n)$, where c_n are the data bits modulated onto the subcarriers and h_n are the channel values at f_n . We want to extract this channel information to compute

the Channel Impulse Response $CIR(\tau)$. Since the preamble bits c_n are known, we can compensate for c_n and compute the $CIR(\tau)$ by taking an IFFT of the channel values h_n . However, this requires capturing the entire bandwidth of the 5G OFDM signal. Our goal is to recover the CIR using a narrowbandwith. To do so, we leverage the MEMS spike-train filter.

The spike-train filter response is made up of uniformly spaced passbands as shown in Fig. 2(b). The spike-train filter serves to sparsify the OFDM symbol by selectively passing subcarriers that fall inside the MEMS passbands, while suppressing all other frequencies. Let the set of frequencies passed by the spike-train be indexed by M . Then, the frequency domain of the signal $\tilde{X}(f)$ ($\tilde{x}(t)$ in the time domain) after passing through the spike-train filter will be $\tilde{X}(f) = \sum_{i \in M} c_i h_i \delta(f - f_i)$.

This sparse spectrum is shown in Fig. 2(b). Next, the IoT receiver subsamples the signal $\tilde{x}(t)$ using a low-speed ADC that samples at a rate $R = B/P$, where B is the bandwidth of the transmitted symbol and P is an integer corresponding to the subsampling factor. Let $y(t)$ be the subsampled signal, that is, $y(t) = \tilde{x}(P \times t)$, and let $Y(f)$ be its frequency domain representation. Then $Y(f)$ is an aliased version of $\tilde{X}(f)$:

$$Y(f) = \sum_{i=0}^{P-1} \tilde{X}(f + iR) \quad (2)$$

$Y(f)$ will cover a narrow bandwidth equal to R MHz as depicted in Fig. 2(c). The process of aliasing is as follows. Any frequency f_j , $j \in M$, that falls outside the narrowband of the IoT device, will alias onto the frequency bin \tilde{f}_j inside the narrowband after subsampling, such that $f_j - \tilde{f}_j = z \times R$, where z is some integer. Note that for every f_j , we have a unique \tilde{f}_j . So given the measurement at the aliased frequency \tilde{f}_j , we can potentially recover the channel value h_j at the corresponding unaliased frequency f_j .

However, recovering these channel values from the aliased spectrum is non-trivial because multiple of the frequency subcarriers passed by the spike-train filter may collide by aliasing on top of each other and summing up. This is unfavorable since now we are unable to extract the channel values for any of the colliding frequencies. Past work addresses this by leveraging multiple co-prime subsampling factors, which ensures that the same frequencies don't collide repeatedly.

Unfortunately, we do not have such flexibility to choose any sub-sampling factor here. This is because in order to recover the channel value h_j from the aliased frequency \tilde{f}_j , we need to ensure that the complex scaling factor $c_j \times h_j$ encoded on subcarrier f_j remains preserved upon aliasing. This is crucial because the wireless channel information is contained inside this scaling factor. The following lemma states the condition that ensures this:

Lemma 5.1. *For a sub-sampling factor P and N OFDM subcarriers, the complex valued scaling factors for each subcarrier will be preserved upon aliasing if $N = z \times P$, for some integer z , given the aliasing results in no collisions.*

The proof for the above lemma is in Appendix A. Thus, to be able to recover channel values, we are restricted to subsample the signal by an integer factor of N . Further, since the OFDM subcarriers in the 5G standard are set to powers of 2, we can only subsample the wideband signal by powers of 2.

Due to this lack of choice in subsampling factors, we instead shift our focus on designing the spike-train filter such that the frequencies passed by the filter do not collide upon aliasing. We achieve this by leveraging the structured periodic sparsity of the spike-train, and design a filter that ensures no collisions for the given subsampling factor P .

Doing so significantly simplifies our recovery algorithm. In particular, given that (1) the frequency response of the spike-train filter and its collision-free aliasing patterns are known, and that (2) the scaling factors at the frequency subcarriers remain preserved upon aliasing, we can now simply rearrange the frequencies in $Y(f)$ to their corresponding unaliased frequency positions as shown in Fig. 2(d). Further, we can extract the channel values at these unaliased frequencies by dividing the complex scaling factor $c_j \times h_j$ by the known preamble bit c_j . Thus, by leveraging the spike-train filter, ISLA is able to extract wideband channel values on a narrow band IoT device. Next, we discuss the design parameters of the spike-train filter that ensures no collisions.

Spike-Train Filter Design: We explain the spike-train filter design with a specific example, shown in Fig. 3(a). Let the wideband transmitted OFDM signal (B MHz bandwidth) be comprised of 32 frequency subcarriers, indexed from -16 to 15, with 0 denoting the carrier frequency bin. From Lemma 5.1, we want the subsampling factor P to divide $N = 32$. So we choose $P = 4$, that is, the IoT receiver subsamples the signal by $4 \times$. This implies that the IoT receiver is only able to capture $\frac{N}{P} = 8$ frequency bins centered around the carrier frequency as shown by the shaded region in Fig. 3(a). Let this narrow band set of frequencies be denoted as f_{NB} .

Recall that when you subsample a B MHz signal by $P \times$, then all frequency subcarriers spaced by $R = \frac{B}{P}$ MHz will alias onto the same frequency bin in the narrow band spectrum. Here, this translates into all frequencies spaced by 8 subcarriers aliasing onto the same narrowband bin. This is depicted in Fig. 3(a) through the color coding scheme. For instance, the subcarriers at $\{-9, -1, 7, 15\}$ (represented as purple colored) would all appear at frequency bin -1 in the narrow band spectrum upon aliasing. For a given subcarrier k in the narrow band spectrum, that is, $k \in \{-4, \dots, 3\}$, let us denote the set of subcarriers that would alias into k as I_k . So we have $I_{-1} = \{-9, -1, 7, 15\}$.

The spike-train filter will selectively pass frequency subcarriers in the wideband OFDM signal, which after aliasing can be recovered from the narrow band signal at the receiver. Let the set of frequency subcarriers passed by the spike-train filter be denoted by f_M , where $M \in [-15, \dots, 16]$. We want the following conditions to hold:

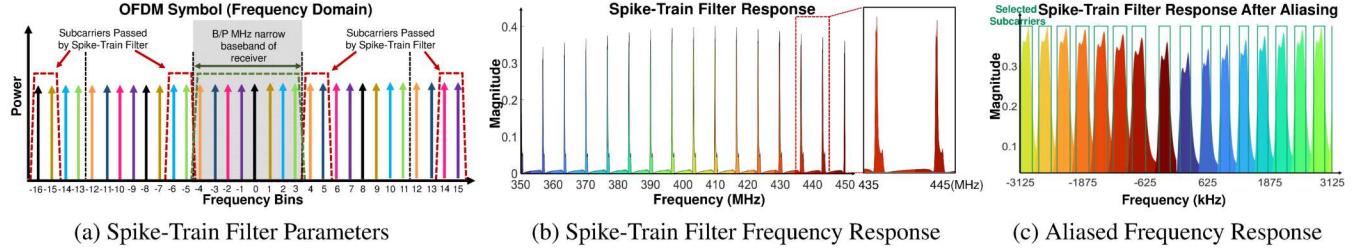


Figure 3: (a) MEMS Filter Parameters that ensure zero collisions while recovering maximum channel information. (b) Frequency response of MEMS spike-train filter. (c) Aliasing pattern of spike-train filter frequency response.

1. *No Collisions:* To ensure that we can successfully recover the wideband channels, no two subcarriers in f_M should alias and collide in the same narrowband frequency bin upon subsampling. To achieve this, the spike-train filter must satisfy: *For any set I_k where $k \in \{-4, \dots, 3\}$, f_M must contain at most one subcarrier from I_k .*
2. *Extract Maximum Possible Channel Values:* Given that the narrowband spectrum spans 8 frequency subcarriers, this means that the receiver can successfully recover at most 8 channel values after subsampling. In the presence of noise, we want to recover as many channel measurements as possible for robustness. Hence, every narrowband subcarrier in f_{NB} should yield one channel measurement from the wideband signal. This translates to: *For any set I_k where $k \in \{-4, \dots, 3\}$, f_M must contain at least one frequency subcarrier from I_k .*
3. *Span the Wideband OFDM symbol:* To retain the high ToF resolution, we want the set of frequencies in f_M to span the entire wideband signal.

The above conditions can be met leveraging the structured sparsity in the spike-train filter response. Specifically, we can design three key parameters of the spike-train filter: (1) spacing between consecutive spikes ΔF , (2) width of the spikes ΔS , and (3) the starting frequency subcarrier f_M^0 in the spike-train, to follow Lemma 5.2. We prove in Appendix A that such a filter response satisfies the above conditions.

Lemma 5.2. Consider an OFDM symbol with N frequency subcarriers, indexed as $\{f_{-\frac{N}{2}}, \dots, 0, \dots, f_{\frac{N}{2}-1}\}$ with inter-frequency spacing of Δf , and a narrowband receiver that subsamples by $P \times$. If P^2 divides N , then the ideal filter parameters that meet all three requirements are: (1) $f_M^0 = f_{-\frac{N}{2}}$, (2) $(\frac{N}{P^2} - 1) \times \Delta f < \Delta S < \frac{N}{P^2} \times \Delta f$, and (3) $\Delta F = \frac{N}{P}(1 + \frac{1}{P}) \times \Delta f$.

Furthermore, we can achieve the required filter response by designing the topology of the MEMS resonators, which we explain in more details in Appendix B.

In Fig. 3(a), we show the ideal frequency response of the spike-train filter designed with the above parameters as the red dotted line. In theory, such a filter should allow us to leverage all f_{NB} subcarriers to recover the wideband channel measurements from the aliased signal. However, in practice,

MEMS spike-train filters are non-ideal i.e., the roll-off of the passband boundaries are not as sharp as perfect rectangular functions, the spikes are not perfectly equally spaced, and the passband widths are not identical. These imperfections can be observed in the frequency response shown in Fig. 3(b). As a result of these non-idealities, there will still be collisions at the boundary regions of the spikes after aliasing, as shown in Fig. 3(c). To avoid collisions from polluting our CIR estimates, we only consider the subcarriers that do not collide as shown in Fig. 3(c). However, this results in non-uniform sampling of the OFDM subcarriers across the wideband channel. In sec. 6, we show how to leverage ISLA’s super-resolution algorithm to recover high resolution CIR estimates from these non-uniform channel measurements.

Tradeoff Between Range and Resolution: Recall from section 3 that the resolution in ToF depends on bandwidth, whereas the maximum unambiguous ToF (range) depends on the inter-frequency spacing between channel measurements. In the 5G OFDM signal with bandwidth $B = 100$ MHz and subcarrier spacing $\Delta f = 60$ kHz, ISLA is able to retain the high ToF resolution of 10 ns (3 m) by collecting wideband channel measurements that span the entire 100 MHz. However, in doing so, the frequency spacing between the channel measurements in ISLA increases, thus reducing the maximum ToF range. Specifically, the frequency spacing increases by $P = 16 \times$ in ISLA, thus reducing the maximum range from 5 km to 312 meters. This is an issue since now it becomes difficult to identify the LoS path from the CIR for localization. You could have the case where the LoS path is at 200 meters but a reflected path at 400 meters aliases and appears at the bin corresponding to 88 meters in the CIR. Thus, you cannot simply pick the first peak as LoS.

To address this, ISLA combines the wideband channel measurements from the spike-train filter, h_M , with the narrowband channel measurements h_{NB} collected at the subcarriers f_{NB} , and formulates a joint optimization with both these channels to estimate the CIR. Since the narrowband channel measurements h_{NB} retain the same subcarrier spacing of $\Delta f = 60$ kHz, it increases the effective maximum ToF range back to 5 km, thus resolving the LoS ambiguity in the CIR.

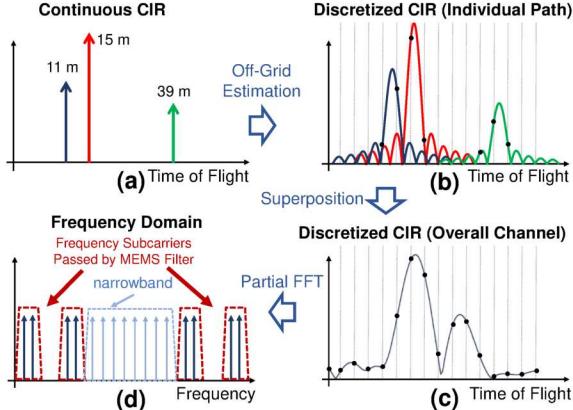


Figure 4: Signal paths to measured channel forward function

6 Super-Resolution CIR Estimation

Here we describe our super-resolution algorithm that can retrieve high resolution ToF estimates τ_l 's along with the associated complex attenuations a_l for the L multipath components in the channel. As discussed in Sec. 5, the IoT device can recover channel measurements $h_{tot} = h_M \cup h_{NB}$ at the subcarriers $f_{tot} = f_M \cup f_{NB}$ where f_M are recovered from the spike-train filter and f_{NB} without the filter. Since these channel values are sampled at non-uniformly spaced frequencies, we cannot apply standard super-resolution algorithms like MUSIC with spatial smoothing [21, 44] as they require uniform measurements. Instead, we optimize for the channel impulse response in the continuous time domain by leveraging an off-grid estimation technique that can estimate high resolution ToF values from the channel information.

We begin by framing this as an inverse problem. We start by modeling the forward operator \mathcal{F} : $h_{tot} = \mathcal{F}(\tau_1, \dots, \tau_L, a_1, \dots, a_L)$, which maps physical path parameters to the wireless channel. \mathcal{F} comprises of the following distinct transformations, as illustrated in Fig. 4:

(1) CIR in Continuous Domain: (Fig. 4(a)) Given path parameters $\{\tau_1, \dots, \tau_L, a_1, \dots, a_L\}$, the continuous domain CIR can be written as: $CIR_{cont} = \sum_{l=1}^L a_l \delta(\tau - \tau_l)$, with each path represented as an impulse positioned at its respective ToF τ_l , and scaled by its complex attenuation a_l .

(2) Off-Grid Estimation: (Fig. 4(b)) The OFDM symbol spans a bandwidth B MHz and comprises of N subcarriers. Due to this discretization and truncation in the frequency domain, the observed CIR at the receiver will also be discretized, and computed on the grid defined by τ_g , where $\tau_g = \left\{ \frac{0}{B}, \dots, \frac{(N-1)}{B} \right\}$. However, as with most natural signals, the ToFs of the physical propagation paths τ_l will rarely align with this discretized τ_g grid, that is, the τ_l 's will lie at an off-grid position. As a result, the leakage from the continuous off-grid CIR component from path l to the discrete CIR grid positions at τ_g can be computed as $CIR^l(\tau_g) = a_l \psi_N(\tau_g - \tau_l)$,

where ψ_N is the discretized sinc function defined as:

$$\psi_N(\tau) = \frac{\sin(\pi\tau)}{\sin(\frac{\pi\tau}{N})} \exp\left(-\pi j\left(\frac{N-1}{N}\right)\tau\right) \quad (3)$$

(3) Superposition: (Fig. 4(c)) With multiple propagation paths in the channel, the net observed CIR at the receiver is the sum of the CIR profiles contributed by each propagation path: $CIR^{net}(\tau_g) = \sum_{l=1}^L a_l \psi_N(\tau_g - \tau_l)$.

(4) Discrete Fourier Transform: (Fig. 4(d)) Finally, the channel h_{tot} can be computed by sampling the corresponding frequencies f_{tot} from the DFT of the superposed CIR. Let us denote the $N \times N$ Fourier matrix as \mathbf{F}_N , and let \mathbf{V} be the matrix that chooses the rows corresponding to f_{tot} from \mathbf{F}_N . Then we have: $h_{tot} = \mathbf{V} \mathbf{F}_N CIR^{net}$ where CIR^{net} is a $N \times 1$ dimension vector.

Putting the above four transformations together, the forward operator \mathcal{F} can be expressed as:

$$h_{tot} = \mathcal{F}(\{\tau_l, a_l\}_{l=1}^L) = \mathbf{V} \mathbf{F}_N \Psi \vec{a} \quad (4)$$

where Ψ is a $N \times L$ matrix with $\Psi_{i,j} = \psi_N(\tau_i - \tau_j)$, and \vec{a} is a $L \times 1$ vector comprising the complex attenuations a_l for each path. Now that we have the forward operator, the inverse problem to retrieve the path parameters from observed channel vector h'_{tot} can be formulated as a L-2 minimization:

$$\{\tau_l^*, a_l^*\}_{l=1}^L = \arg \min_{\tau_1, \dots, \tau_L, a_1, \dots, a_L} \|h'_{tot} - \mathbf{V} \mathbf{F}_N \Psi \vec{a}\|^2 \quad (5)$$

Solving the Optimization: Note that if we are given Ψ , then Eq. 5 becomes a linear optimization problem in \vec{a} . Thus, given Ψ , the closed form solution for \vec{a} that minimizes Eq. 5 is $\vec{a} = (\mathbf{V} \mathbf{F}_N \Psi)^{\dagger} h'_{tot}$, where \dagger represents the pseudo-inverse. Thus the objective function in Eq. 5 can be rewritten as:

$$\begin{aligned} \{\tau_l^*\}_{l=1}^L &= \arg \min_{\tau_1, \dots, \tau_L} \|h'_{tot} - \mathbf{V} \mathbf{F}_N \Psi (\mathbf{V} \mathbf{F}_N \Psi)^{\dagger} h'_{tot}\|^2 \\ \text{s.t. } \tau_l &\geq 0 \quad \forall l \in \{1, 2, \dots, L\} \end{aligned} \quad (6)$$

The objective function is now reduced to just the ToF variables τ_l 's. This optimization problem is non-convex and constrained, and we use the well-known interior-point method to solve this [8]. For the initialization point to the optimization algorithm, we use approximate ToF values from the CIR computed by taking the inverse FFT of the observed channel h'_{tot} . While these ToF estimates are distorted by the discretization and superpositioning artifacts described previously, it gives a good starting point for the optimization.

Also, note that the number of paths N in the wireless channel is not known a priori. As we keep increasing the number of paths N that the algorithm is initialized with, it keeps finding a better and better fit to the channel data, and after a point, starts overfitting to the noise. In order to avoid overfitting and yet yield accurate estimates for the path parameters, we run the optimization problem multiple times, each time increasing the number of paths it is initialized with by 1. We terminate the algorithm when the decrease in the value of the objective function falls below some threshold ϵ , and set the current value of N to be the number of paths in the channel.

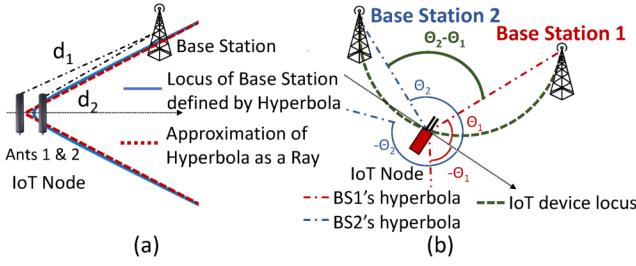


Figure 5: ISLA’s Localization Algorithm

7 ISLA’s Localization Algorithm

The above off-grid estimation algorithm gives us highly precise ToF estimates for the propagation paths. However, since the 5G base stations are not time synchronized with the IoT device, there is going to be an offset between the sampling clocks in their RF chains. As a result, the measured ToF at the IoT node also includes delays from the sampling time offset (STO) between the different base stations and the IoT node, and hence cannot provide accurate distance estimates.

To address this, *ISLA* leverages two antennas on the IoT node to compute the differential ToF rather than the absolute. The key idea here is that while the absolute ToF measurements are corrupted by synchronization offsets, these offsets are constant across the two antennas on the IoT node. Hence, the offsets can be eliminated by differencing the two measurements. Let the ToF values to the two antennas be τ_1 and τ_2 , and their corresponding distances be d_1 and d_2 , as denoted in Fig. 5(a). Then the locus of the base station from the IoT device’s frame of reference is a hyperbola with the two antennas being the foci, and the difference in distances to the two foci equaling $d_2 - d_1$. At large distances, this hyperbola can be approximated as two rays along the asymptotes of the hyperbola, depicted by the red dashed lines in Fig. 5(a).

By overhearing packets from different base stations, the IoT device can infer the locus of each base station to lie on approximated rays originating from the IoT device’s location. This is shown in Fig. 5(b), where base station 1 can lie on the rays at angles θ_1 or $-\theta_1$, and similarly the base station 2 can lie on the rays at angles θ_2 or $-\theta_2$. Both θ and $-\theta$ are possible since there is the ambiguity that the signal might have arrived from the front or the back of the device. Given this, we can see that the angle subtended by the two base stations at the location of the IoT device will be $\|\theta_2 - \theta_1\|$, and this is going to be constant irrespective of the orientation of the IoT node. (There is ambiguity in that the angle subtended can also be $\|\theta_2 + \theta_1\|$, and we will address this shortly).

Given the angle subtended by the base stations and the known locations of the base stations, according to the Inscribed Angle Theorem, we can determine the locus of the IoT device to lie on the arc of a circle, where the line segment connecting the two base stations is the chord and the corresponding inscribed angle is equal to the angle subtended by the base stations. This is illustrated in Fig. 5(b) as the green dashed arc. Leveraging different pairs of base stations, *ISLA*

can draw multiple such arcs and the intersection points of these arcs will give us the IoT device’s location.

Sources of Ambiguity: There are some sources of ambiguity that need to be resolved. First, the angle subtended by the two base stations in Fig. 5(b) could also be $\|\theta_2 + \theta_1\|$, and second, the arc drawn with the base stations at the end points could also be pointing towards the north rather than south, as depicted in Fig. 5(b). These ambiguities can be resolved easily by leveraging 4 base stations as anchor points. Keeping one base station common, we have three base station pairs which yields three unique arcs. Only the right configurations of angles subtended and arcs drawn will give us a common intersection point for all three arcs. *ISLA*’s localization algorithm tries all configurations and picks the one where all arcs coincide at the same point.

8 Integrating ISLA with 5G-NR Standard

Similar to the LTE standard, the 5G-NR packet consists of 10 subframes, each of duration 1 ms [28]. To allow for coherent packet demodulation, the 5G frame appends known preamble bits on each subframe which enables channel estimation and correction across the entire bandwidth of the 5G channel. Additionally, in the first subframe of the packet, the base station also includes all information required by devices to associate with the network, which comprises of the synchronization signals (PSS and SSS frames) for CFO correction and frame timing, and the Base Station ID. To allow every device in the network to receive this critical information, it is always encoded in the narrowest supported bandwidth of the wideband packet, which is 4.32 MHz in the 5G standard [28].

ISLA’s hardware circuit, discussed in Section 9, is designed such that it can switch between capturing the 6.25 MHz narrowband spectrum, or the wideband spectrum via the spike-train filter. *ISLA* begins by capturing the first subframe of the 5G packet through its narrowband RF path, and extracts the synchronization frames and base station ID encoded in the narrowband subcarriers of the wideband packet. Using publicly available databases like [1], *ISLA* can retrieve the location of the Base Station given its ID. The synchronization frames help eliminate coarse CFO and SFO. From the subsequent subframes, *ISLA* first estimates the narrowband channel, and then switches to the RF path with the spike-train filter to sense wideband channel. Note that *ISLA* does not need to meet tight timing constraints to switch since each subframe lasts 1 ms and there are multiple such subframes in each packet that can be leveraged for channel estimation. Thus, *ISLA* can simply skip a subframe while switching.

However, because *ISLA* captures the narrowband channel and wideband channel from different subframes, there is going to be an additional phase accumulation between the two measurements due to residual CFO. To address this, we slightly modify Eq.6, and the detailed description for this modification is presented in Appendix C.

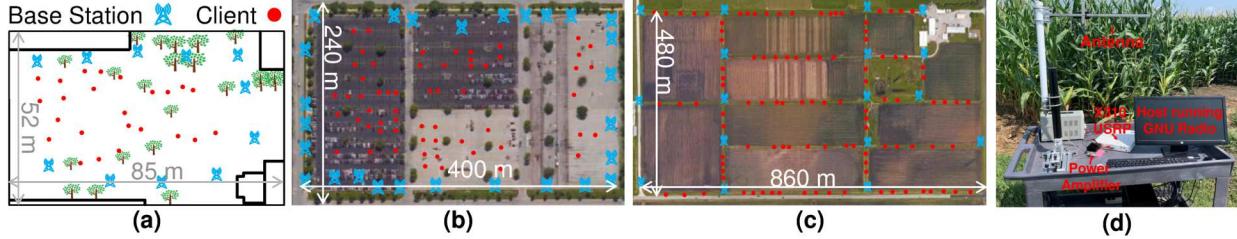


Figure 6: Ourdoor Experiment Testbeds: (a) Campus testbed surrounded by buildings. (b) Parking lot testbed. (c) Agricultural farm testbed. (d) Prototype base station in the agricultural farm testbed.

9 System Implementation

System Design: We have built a prototype *ISLA* device by combining our MEMS spike-train filter with commodity, off-the-shelf, low-power components. Figure 7(a) shows the circuit diagram, and Fig. 7(b) shows the actually prototype. It receives ambient 5G transmissions with two antennas followed by identical RF chains. Depending on whether the IoT devices wants to receive the full 100 MHz spectrum using the spike-train filter or the narrowband spectrum, the RF chains can switch between two paths: (1) the received wideband spectrum first be filtered by the MEMS spike-train filter, and then down-converted and sampled without using the anti-aliasing filter. (2) the MEMS spike-train filter is bypassed but the down-converted signal will first go through an anti-aliasing filter before sampling. We select between the two paths using RF switches controlled by a single microcontroller.

Implementation: We fabricated a MEMS spike-train filter at 400 MHz center frequency. However, due to the strong interference from the amateur radios in this band, we were not able to run experiments outdoor using this filter. Hence, the above prototype was only used indoors. In the outdoor experiments, we transmitted in a vacant 100 MHz wide spectrum between 950 and 1050 MHz, and we emulate the IoT radio front-end described above with the MEMS spike-train filters in digital using an X310 USRP software-defined radio (SDR). We would like to note that in practical deployments we do not expect interference to play a major issue since *ISLA* will be deployed in the proprietary frequency bands licensed by cellular companies, which in turn will have limited interference.

The X310 SDR has two identical RF chains, and can sample the full 100 MHz bandwidth with UBX160 daughterboards. To emulate the MEMS spike-train in digital, we first measure the spike-train filter frequency response once using a vector network analyzer (VNA), and we apply this filter frequency response to the received signals sampled at 100 MHz. Then, we downsample the filtered signal by simply keeping every 16th sample. This is equivalent to filtering the RF signal in analog and sample it below the Nyquist sampling rate. We also used a bandpass filters between the antenna and SDRs to remove out-of-band interferences and synchronized the two RF chains in time and phase through the GNU Radio Python API. In section 10.3, we present microbenchmarks demonstrating the equivalence between applying the filter in

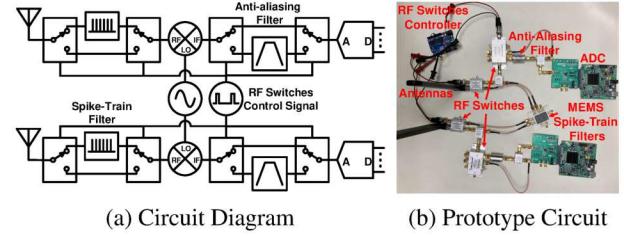


Figure 7: ISLA Prototype Circuit

digital and the above hardware prototype.

Testbed: Additionally, we also built 5G base station TX prototypes to transmit ambient 5G communication signals. As shown in Fig 6(d), the base station prototype consists an X310 USRP SDR with a UBX160 daughterboard, a 9 dBi Yagi directional antenna, and an RF Bay MPA-22-30 30 dB power amplifier. The base stations transmit 100 MHz OFDM packets. Using five base station prototypes, we created three testbeds with different dimensions and at different locations to conduct our experiments. Figure 6 shows the satellite images of our testbeds with the base stations and clients locations marked. The first testbed is 85 m long and 52 m wide on a university campus, surrounded by buildings on all sides. We designated 11 basestation locations in this testbed and chose five of them for each experiment. The second testbed is a 400 m by 240 m parking lot with 27 base station locations. The third testbed is at a 102 acre farmland with 860 m length 480 m width. We selected five out of the 17 potential locations to place the base stations in each experiment. For ground truth locations, we used differential GPS RTK with real-time RTCM correction data, which provides centimeter-level positioning accuracy.

10 Experimental Evaluation

10.1 Baselines

- (1) *Spot-Fi*: [21] proposes a 2D MUSIC algorithm with spatial smoothing, which can localize clients by separating the multi-path components jointly along the ToF and AoA domains.
- (2) *mD-Track*: [43] separates propagation paths by leveraging multiple dimensions of the wireless signal (ToF, AoA, AoD and Doppler), and proposes an iterative algorithm that goes through multiple rounds of error computation and path re-estimation. In our experimental setup, leveraging the AoD and Doppler dimensions provides little benefit since the base

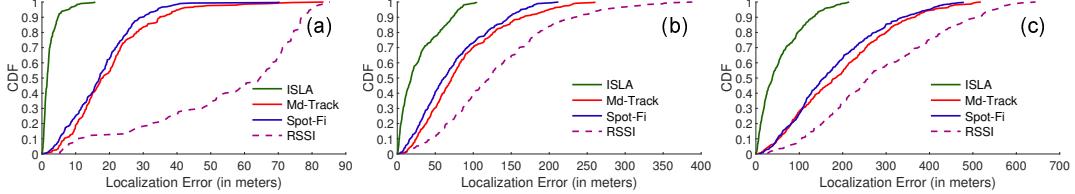


Figure 8: ISLA’s localization accuracy compared against baselines across different testbeds: (a) Campus (b) Parking lot (c) Farm.

station is equipped with a single antenna and the IoT device does not have high mobility relative to the base station. Note that, systems like Spot-Fi and mD-Track were not designed for ambient localization, and thus need to be adapted here. Specifically, we leverage the ToF estimates provided by these baselines for the LoS path, and in turn self-localize the client by computing the relative ToF, as described in Section 7. (3) *RSSI*: Past work leverages RSSI measurements to localize clients in outdoor cellular networks, by either using approximate path loss models for trilateration, or by using the known locations of nearby cells as coarse estimates. We implemented one recent RSSI baseline [9].

(4) *Spike-train filter-adapted baselines*: To provide a fair comparison against *ISLA*, we modify Spot-Fi and mD-Track to leverage the spike-train filter and utilize the wideband channel measurements for localization. It is non-trivial to adapt Spot-Fi for the spike-train filter since the spatial smoothing technique used in Spot-Fi requires uniformly spaced channel measurements across frequency, whereas the spike-train filter samples the OFDM frequency bins non-uniformly. To address this, we restructure the spatial smoothing subarray from [21] that allows Spot-Fi to be applied across the non-uniform frequencies sampled by the spike-train filter.

10.2 Results

Unless otherwise specified, for all results, we utilize 5 randomly chosen base stations as the anchor points.

A. Localization Accuracy Comparison against Baselines: We compare *ISLA*’s localization against the baselines in Fig. 8. Note that, while *ISLA* is designed specifically to leverage the wideband channel sensed by the MEMS filter, the baselines are implemented without modification and thus utilize only the narrowband channel for localization.

From Fig. 8, *ISLA* achieves a median localization accuracy of 1.58 meters in the campus testbed, 17.6 meters in the parking lot testbed, and 37.8 meters in the farm testbed. Across the same three testbeds, Spot-Fi achieves median accuracies of 17.05 meters, 61.2 meters and 156.6 meters, whereas mD-Track achieves 18.11 meters, 71.8 meters, and 183.1 meters respectively. Thus, *ISLA* improves the localization accuracy over Spot-Fi and mD-track by $\sim 11\times$ in the campus testbed, and by $\sim 4\times$ in the parking lot and farm. *ISLA* is able to achieve such high gains since it leverages the spike-train filter to sense wideband channel on the narrowband device, which allows for much higher resolution compared to the baselines

operating solely in the narrowband. Further, the localization improvement over the narrowband baselines is most significant in the campus testbed, since it has the most multipath from surrounding buildings, and thus ToF resolution is critical to separate out the LoS path from reflections.

Lastly, the RSSI baseline achieves median accuracies of 64.54 meters, 120.7 meters, and 260.8 meters respectively across the three testbeds. RSSI based methods generally have poor performance, as they tend to oversimplify path loss models that map RSSI values to distance, which does not hold for real world multipath channels.

B. Comparison against Spike-train-adapted Baselines: Next, we evaluate how leveraging the spike-train filter would benefit the performance of our narrowband baselines. Fig. 9 shows the CDF of localization accuracy comparing *ISLA* against the modified baselines that utilize the wideband channel from the spike-train filter. The RSSI baseline is not included here since its localization performance does not depend on bandwidth. Compared to its narrowband implementation, Spot-Fi’s median accuracy improves to 11.08 meters in the Campus testbed, 49.07 meters in the Parking Lot, and 137.76 meters in the farm. Similarly, mD-Track’s median performance improves to 15.48 meters, 51.45 meters and 103.78 meters in the three testbeds respectively. Thus, Spot-Fi and mD-Track see improvements in localization accuracy by up to 54% and 76% respectively. This shows that other localization techniques can also benefit from the wide-band channel sensing capabilities enabled by the spike-train filter.

Additionally, Fig. 9 shows that given the same channel information, *ISLA*’s off-grid CIR estimation algorithm is able to better resolve and estimate the relative ToF compared to Spot-Fi and mD-Track. This is because these baselines were designed to leverage multiple information dimensions to separate out the multipath components, with both baselines leveraging 3 or more antennas for separation in the AoA domain, and mD-Track further using the additional dimensions of Doppler and AoD as well. In contrast, here the IoT device has to separate out multipath in the ToF domain alone, and *ISLA* is able to achieve very accurate localization owing to its off-grid estimation algorithm.

C. ISLA Leveraging Different Amounts of Spectrum: In this experiment, we compare *ISLA*’s localization algorithm applied across three different amounts of spectrum utilization — (1) *ISLA* applied only to the wideband sparse channel sensed by the spike-train filter (without combining with narrowband channel), (2) *ISLA* applied only to the narrowband channel of

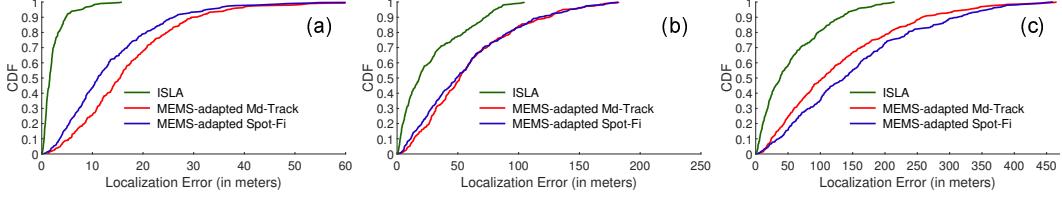


Figure 9: ISLA’s localization accuracy compared against MEMS filter adapted baselines at: (a) Campus (b) Parking lot (c) Farm.

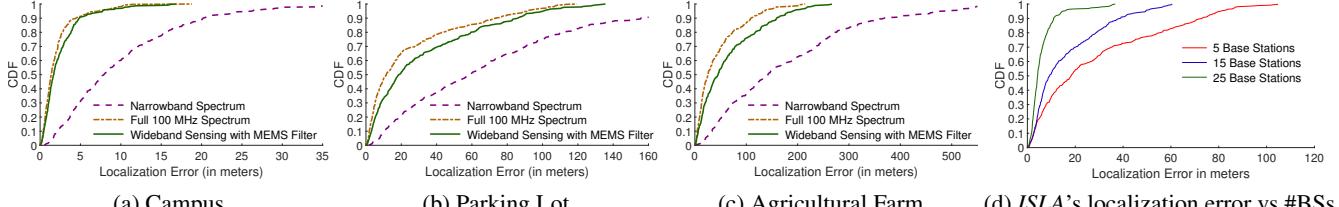


Figure 10: (a-c) Comparison of ISLA’s localization accuracy when leveraging different amounts of spectrum across all three testbeds. (d) ISLA’s localization error with different number of visible base stations.

IoT device, and (3) ISLA applied across the entire 100 MHz bandwidth of the received 5G signal. Fig. 10 plots the CDF of localization accuracy achieved across the three testbeds.

ISLA applied on the narrowband channel performs the poorest, achieving median accuracies of 7.9 meters, 58.9 meters and 142.52 meters in the campus, parking lot and farm testbeds. In contrast, ISLA along with the spike-train filter can achieve corresponding median accuracies of 1.68 meters, 18.8 meters and 45.04 meters. Thus, ISLA along with spike-train achieves an improvement in localization accuracy of $3.16 \times - 4.7 \times$ compared to ISLA applied in the narrowband spectrum, despite both baselines capturing the same amount of channel measurements. The advantage of spike-train stems from the fact that it enables the narrowband receiver to capture channel measurements that span a much larger bandwidth, which results in much higher ToF resolution.

On the other hand, ISLA’s localization algorithm applied on the full 100 MHz spectrum achieves median accuracies of 1.38 meters, 11.44 meters and 25.8 meters respectively on the three testbeds. Thus, ISLA with the spike-train filter reduces the localization accuracy by only $1.21 \times$, $1.64 \times$, and $1.74 \times$ respectively compared to this upper bound. This demonstrates that the spike-train filter can enable a narrowband device to achieve localization accuracy within a factor of $2 \times$ compared to a broadband receiver, despite the fact that it subsamples the signal by $16 \times$ below Nyquist.

D. Localization with Number of Anchor Base Stations:

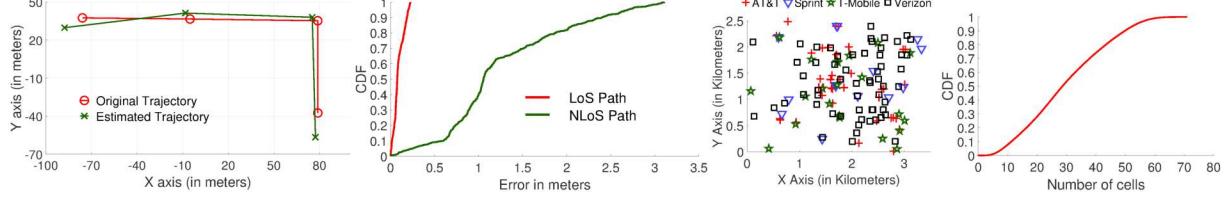
In Fig. 10(d), we compare ISLA’s localization performance with 5, 15 and 25 base stations used as anchor points respectively, in the parking lot testbed. With 5 base stations, ISLA achieves a median accuracy of 17.6 meters, which improves to 9.27 meters with 15 base stations, and 4.26 meters with 25 base stations. This improvement becomes even more significant at the tail, with ISLA achieving 90th percentile accuracy of 73.16 meters with 5 base stations, which improves to 10.9 meters accuracy with 25 base stations at 90th percentile. Thus,

leveraging more base stations can significantly improve the localization accuracy achieved by ISLA.

E. Tracking Objects: We move the IoT device across an L-shaped trajectory (160 meters in length and 85 meters in width) in the parking lot testbed, and collect packet transmissions from the base stations at different points along this trajectory. In this experiment, we pick 7 fixed base stations to utilize as anchor points, and we show the ground truth trajectory and corresponding estimated trajectory by ISLA in Fig. 11(a). As can be observed, ISLA’s high localization accuracy allows to faithfully capture the shape of the ground truth trajectory.

10.3 Microbenchmarks

A. CIR Estimation using Fabricated MEMS Spike-train Filter: To verify the equivalence between our outdoor implementation and using the prototype with the fabricated MEMS spike-train filter at 400 MHz, we conduct indoor experiments at 400 MHz. Specifically, we evaluate the error in reconstructed CIR and estimated ToF values between the prototype with the fabricated filter and ISLA with the digital filter implementation. In Fig. 11(b), we show the CDF of the errors in ToF values (converted to distance (meters)) recovered by the two approaches, for both LoS and NLoS paths. We can see that the position of the LoS path in the CIR estimated from both approaches are very close, with the median error between their estimates being 0.075 meters. The error in the NLoS paths is higher, with a median error of 1.05 meters. However, this will not affect the localization performance between the two since localization only uses the LoS path. This microbenchmark demonstrates that ISLA’s approach of applying the filter and subsampling in digital is equivalent to using the fabricated filter from a localization perspective, and that the results shown in this paper are representative of a fully implemented system.



(a) Tracking Object Trajectory

(b) Fab. MEMS filter vs ISLA's Imp.

(c) 4G BS Density

(d) # of Visible BSs

Figure 11: (a) Using *ISLA* to track object trajectory. (b) ToF difference between *ISLA*'s prototype with fabricated MEMS filter and digitally implemented MEMS filter. (c) Deployment of 4G base stations in the downtown area of a major US city. (d) Number of visible 4G base stations at various downtown locations.

Direction	NW	NE	SE	SW
Median	1.3535 m	1.3544 m	1.3267 m	1.3681 m
Std Dev	0.4948 m	0.6026 m	0.4908 m	0.512 m

Table 1: Invariance of Localization Error to Orientation

B. Density of Deployed Base Stations: In Section 10.2D, we have shown that *ISLA*'s localization accuracy increases substantially as we use more anchor base stations. Here, we study the distribution of how many base stations can the client overhear at a given location. Using publicly available databases [1], we retrieved the locations of 4G LTE base stations belonging to 4 major carriers in the United States. We chose 4G LTE for this analysis since 5G deployment is still in its nascent stage in the USA, but we expect the target coverage for 5G networks to exceed the 4G deployment.

In Fig. 11(c), we show the scatter plot of the 4G base stations located in the downtown area of a major metropolitan city in the USA. Using the cell coverage information provided in [1] for the different base stations, in Fig. 11(d), we plot the CDF of the number of base stations that the client can overhear at different locations on the map. We can see that at the 10th percentile, the number of visible base stations is 11, thus implying that less than 10% of client locations see less than 11 base stations. Further, the median number of base stations visible to the client is 29. This demonstrates that the cellular deployment is dense enough to allow many anchor points, which in turn can achieve high localization accuracy.

C. Invariance to Orientation: Here, we demonstrate that the localization performance is independent of the orientation of the IoT device. This is because the arcs that define the locus of the IoT node, depend only on the angle subtended by the base stations at the IoT device's location, which is invariant to device rotation. At a given location in our campus testbed, we orient the IoT device along 4 different directions and perform 100 localization experiments at each orientation. From Table 1, we can see that the median and standard deviation in localization error is almost the same across the 4 orientations, thus demonstrating invariance to orientation.

11 Limitations and Discussion

- **Power Footprint:** To enable ambient localization, *ISLA* leverages a second antenna and RF chain, which increases the power footprint of the IoT device. However, we would like to note that the power overhead of an additional RF

chain is going to be lower than that of a GPS module, which is the likely alternative for localization. This is because the additional RF chain on the IoT device is going to operate in the narrowband with very low sampling rates, whereas GPS incurs high operational power since it needs to receive and correlate long sequences to get the signal power above the noise floor for GPS lock acquisition. Hence, while *ISLA*'s design does lead to an increased power footprint, it is still a better alternative compared to GPS.

- **Loss of SNR:** Since the MEMS spike-train filter is a passive device, the signal suffers from insertion loss when passed through the filter, thus resulting in loss of SNR. This is further exacerbated by the fact that in practice, the out-of-band rejection of the spike train filter is finite, which results in further loss of SNR. It is possible to reduce the impact of this SNR loss at the circuit level by improving impedance matching and the isolation between input and output ports. We can also compensate for the SNR loss by averaging the channel measurements across multiple OFDM symbols.
- **Line-of-sight:** Similar to many localization systems, *ISLA* assumes the availability of line-of-sight (LoS) paths to the base stations which might not hold under occlusion. This, however, can be addressed by potentially selecting a subset of base stations with LoS paths using similar techniques demonstrated in [21]. With the dense deployment of 5G base stations, we expect a significant subset of base stations to have LoS path to the node.
- **Fast Mobility:** The current design of *ISLA* is not suitable for highly dynamic applications with fast mobility such as tracking cars. This is because the localization algorithm must receive wideband 5G packets from 4 or more base stations before it can self-localize.
- **Multiple Providers:** *ISLA* can benefit from capturing signals from multiple different providers since the IoT node does not need to associate with the base stations. However, different providers operate in different frequency bands which would require different spike-train filters. This could potentially be addressed by having multiple filters and switching between them similar to our design in sec. 9.

Acknowledgements: We thank our shepherd, Vyas Sekar, and the anonymous reviewers for their feedback and comments. We also thank Steffen Link for his help with fabricating the hardware for this project. This work is funded in part by NSF award numbers: 1750725 and 1824320.

References

- [1] Cell Mapper cell tower locations. <https://www.cellmapper.net>. Accessed: Mon, Sep 13, 2021.
- [2] 3GPP. Study on narrow-band internet of things (NB-IoT) / enhanced machine type communication (eMTC) support for non-terrestrial networks (NTN). Technical Report (TR) 36.763, 3rd Generation Partnership Project (3GPP), 06 2021.
- [3] Godfrey Anuga Akpakwu, Bruno J Silva, Gerhard P Hancke, and Adnan M Abu-Mahfouz. A survey on 5g networks for the internet of things: Communication technologies and challenges. *IEEE access*, 6:3619–3647, 2017.
- [4] Heba Aly and Moustafa Youssef. Dejavu: an accurate energy-efficient outdoor localization system. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 154–163, 2013.
- [5] Atul Bansal, Akshay Gadre, Vaibhav Singh, Anthony Rowe, Bob Iannucci, and Swarun Kumar. Owl: Accurate lora localization using the tv whitespaces. In *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (co-located with CPS-IoT Week 2021)*, pages 148–162, 2021.
- [6] Sujitra Boonsriwai and Anya Apavatjrut. Indoor wifi localization on mobile devices. In *2013 10th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, pages 1–5. IEEE, 2013.
- [7] Mathieu Bouet and Aldri L Dos Santos. Rfid tags: Positioning principles and localization techniques. In *2008 1st IFIP Wireless Days*, pages 1–5. IEEE, 2008.
- [8] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.
- [9] Rizanne Elbakly and Moustafa Youssef. Crescendo: An infrastructure-free ubiquitous cellular network-based localization system. In *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2019.
- [10] Sinan Gezici and Zafer Sahinoglu. Uwb geolocation techniques for ieee 802.15.4a personal area networks. *MERL Technical report*, 2004.
- [11] Songbin Gong, Yong-Ha Song, Tomas Manzaneque, Ruochen Lu, Yansong Yang, and Ali Kourani. Lithium niobate mems devices and subsystems for radio frequency signal processing. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 45–48. IEEE, 2017.
- [12] Junfeng Guan, Jitian Zhang, Ruochen Lu, Hyungjoo Seo, Jin Zhou, Songbin Gong, and Haitham Hassanieh. Efficient wideband spectrum sensing using MEMS acoustic resonators. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 809–825. USENIX Association, April 2021.
- [13] Fredrik Gustafsson and Fredrik Gunnarsson. Mobile positioning using wireless networks: possibilities and fundamental limitations based on available wireless network measurements. *IEEE Signal processing magazine*, 22(4):41–53, 2005.
- [14] Ismail Guvenc and Chia-Chin Chong. A survey on toa based wireless localization and nlos mitigation techniques. *IEEE Communications Surveys & Tutorials*, 11(3):107–124, 2009.
- [15] Yixue Hao, Min Chen, Long Hu, Jeungeun Song, Mojca Volk, and Iztok Humar. Wireless fractal ultra-dense cellular networks. *Sensors*, 17(4):841, 2017.
- [16] Haitham Hassanieh, Lixin Shi, Omid Abari, Ezzeldin Hamed, and Dina Katabi. Ghz-wide sensing and decoding using the sparse fourier transform. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 2256–2264, 2014.
- [17] Mohamed Ibrahim and Moustafa Youssef. Cellsense: An accurate energy-efficient gsm positioning system. *IEEE Transactions on Vehicular Technology*, 61(1):286–296, 2011.
- [18] Mohamed Ibrahim and Moustafa Youssef. A hidden markov model for localization using low-end gsm cell phones. In *2011 IEEE International Conference on Communications (ICC)*, pages 1–5. IEEE, 2011.
- [19] Vikram Iyer, Rajalakshmi Nandakumar, Anran Wang, Sawyer B. Fuller, and Shyamnath Gollakota. Living iot: A flying wireless platform on live insects. In *The 25th Annual International Conference on Mobile Computing and Networking, MobiCom ’19*, 2019.
- [20] Michio Kadota, Shuji Tanaka, Yasuhiro Kuratani, and Tetsuya Kimura. Ultrawide band ladder filter using SH0 plate wave in thin LiNbO₃ plate and its application. In *2014 IEEE International Ultrasonics Symposium*, pages 2031–2034, 2014.
- [21] Manikanta Kotaru, Kiran Joshi, Dinesh Bharadia, and Sachin Katti. Spotfi: Decimeter level localization using wifi. In *Proceedings of the 2015 ACM Conference on*

- Special Interest Group on Data Communication*, pages 269–282, 2015.
- [22] Somansh Kumar and Ashish Jasuja. Air quality monitoring system based on IoT using raspberry pi. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 1341–1346. IEEE, 2017.
- [23] Ruochen Lu, Tomás Manzaneque, Yansong Yang, Jin Zhou, Haitham Hassanieh, and Songbin Gong. Rf filters with periodic passbands for sparse fourier transform-based spectrum sensing. *Journal of Microelectromechanical Systems*, 27(5):931–944, 2018.
- [24] E Manavalan and K Jayakrishna. A review of internet of things (IoT) embedded sustainable supply chain for industry 4.0 requirements. *Computers & Industrial Engineering*, 127:925–953, 2019.
- [25] Andreas Marcaletti, Maurizio Rea, Domenico Giustino, Vincent Lenders, and Aymen Fakhreddine. Filtering noisy 802.11 time-of-flight ranging measurements. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 13–20, 2014.
- [26] Saman Naderiparizi, Yi Zhao, James Youngquist, Alan son P Sample, and Joshua R Smith. Self-localizing battery-free cameras. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 445–449, 2015.
- [27] Rajalakshmi Nandakumar, Vikram Iyer, and Shyamnath Gollakota. 3d localization for sub-centimeter sized devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 108–119, 2018.
- [28] Aymen Omri, Mohammed Shaqfeh, Abdelmohsen Ali, and Hussein Alnuweiri. Synchronization procedure in 5g nr systems. *IEEE Access*, 7:41286–41295, 2019.
- [29] Jeongyeup Paek, Kyu-Han Kim, Jatinder P Singh, and Ramesh Govindan. Energy-efficient positioning for smartphones using cell-id sequence matching. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 293–306, 2011.
- [30] Joan Palacios, Guillermo Bielsa, Paolo Casari, and Joerg Widmer. Communication-driven localization and mapping for millimeter wave networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2402–2410. IEEE, 2018.
- [31] Joan Palacios, Paolo Casari, and Joerg Widmer. Jade: Zero-knowledge device localization and environment mapping for millimeter wave systems. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.
- [32] Anshul Rai, Krishna Kant Chintalapudi, Venkata N Padmanabhan, and Rijurekha Sen. Zee: Zero-effort crowdsourcing for indoor localization. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 293–304, 2012.
- [33] Hamada Rizk, Ahmed Shokry, and Moustafa Youssef. Effectiveness of data augmentation in cellular-based localization using deep learning. In *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2019.
- [34] Hazem Sallouha, Alessandro Chiumento, and Sofie Pollin. Localization in long-range ultra narrow band IoT networks using rssi. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2017.
- [35] Ahmed Shokry, Marwan Torki, and Moustafa Youssef. Deeploc: a ubiquitous accurate and low-overhead outdoor cellular localization system. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 339–348, 2018.
- [36] Y. Song and S. Gong. Wideband spurious-free lithium niobate rf-mems filters. *Journal of Microelectromechanical Systems*, 26(4):820–828, 2017.
- [37] Parvathanathan Subrahmanyam and Amir Farajidana. 5g and beyond: Physical layer guiding principles and realization. *Journal of the Indian Institute of Science*, 100:263–279, 2020.
- [38] Adam Thierer and Andrea Castillo. Projecting the growth and economic impact of the internet of things. *George Mason University, Mercatus Center, June*, 15, 2015.
- [39] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. Farmbeats: An iot platform for data-driven agriculture. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 515–529, 2017.
- [40] Deepak Vasisht, Swarun Kumar, and Dina Katabi. Decimeter-level localization with a single wifi access point. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 165–178, 2016.

- [41] Jue Wang and Dina Katabi. Dude, where's my card? rfid positioning that works with multipath and non-line of sight. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 51–62, 2013.
- [42] Fuxi Wen, Henk Wymeersch, Bile Peng, Wee Peng Tay, Hing Cheung So, and Diange Yang. A survey on 5g massive mimo localization. *Digital Signal Processing*, 94:21–28, 2019.
- [43] Yaxiong Xie, Jie Xiong, Mo Li, and Kyle Jamieson. md-track: Leveraging multi-dimensionality for passive indoor wi-fi tracking. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.
- [44] Jie Xiong and Kyle Jamieson. Arraytrack: A fine-grained indoor location system. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 71–84, 2013.
- [45] Jie Xiong, Karthikeyan Sundaresan, and Kyle Jamieson. Tonetrack: Leveraging frequency-agile radios for time-based indoor wireless localization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 537–549, 2015.
- [46] Chouchang Yang and Huai-Rong Shao. Wifi-based indoor positioning. *IEEE Communications Magazine*, 53(3):150–157, 2015.
- [47] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. Deep learning in mobile and wireless networking: A survey. *IEEE Communications surveys & tutorials*, 21(3):2224–2287, 2019.
- [48] C. Zuo, N. Sinha, and G. Piazza. Very high frequency channel-select mems filters based on self-coupled piezoelectric AlN contour-mode resonators. *Sensors and Actuators A: Physical*, 160(1):132 – 140, 2010.

A Proofs

Here we re-state the lemmas and provide proofs.

Lemma 5.1 *For a sub-sampling factor P and N OFDM subcarriers, the complex valued scaling factors for each subcarrier will be preserved upon aliasing if $N = z \times P$, for some integer z , given the aliasing results in no collisions.*

Proof of lemma 5.1: Assume that $x[n]$ is a discrete signal from 0 to $N - 1$, and we are sub-sampling (or *decimating*) it by a factor of P , meaning $y[n] = X[n \times P]$ for some integer P . Then the Discrete Fourier Transform of $y[n]$, denoted by $\hat{Y}[k]$

is

$$\begin{aligned}\hat{Y}[k] &= \sum_{n=0}^{\lfloor N/P \rfloor - 1} x[nP] e^{-j2\frac{2\pi}{N/P}kn} \\ &= \frac{1}{P} \sum_{n=0}^{N-1} x[n] \sum_{m=0}^{P-1} e^{j\frac{2\pi}{P}mn} e^{-j2\frac{2\pi}{N/P}\frac{kn}{P}} \\ &= \frac{1}{P} \sum_{m=0}^{P-1} \left(\sum_{n=0}^{N-1} x[n] e^{-j(\frac{2\pi}{N}n)(k\frac{N/P}{N/P} - \frac{N}{P}m)} \right).\end{aligned}$$

Now if P divides N , in other words $N = Pz$ for some integer z , the above simplifies to

$$\begin{aligned}\hat{Y}[k] &= \frac{1}{P} \sum_{m=0}^{P-1} \left(\sum_{n=0}^{N-1} x[n] e^{-j(\frac{2\pi}{N}n)(k-zm)} \right) \\ &= \frac{1}{P} \sum_{m=0}^{P-1} \hat{X}[k-zm],\end{aligned}$$

where \hat{X} is the DFT of $x[n]$. This proves that, as long as there is no collision, meaning that there is at most one index m in the above equation for which $\hat{X}[k-zm] \neq 0$, then the complex values of $\hat{X}[k]$ will be fully preserved upon sub-sampling. This proves the lemma.

We also point out that if P does not divide N , then the complex values are *not* preserved. Specifically, if N/P is not a proper integer, $\hat{Y}[k]$ will be in terms of $\hat{X}[k\frac{N/P}{N/P} - \frac{N}{P}m]$ where inside the argument, $k\frac{N/P}{N/P} - \frac{N}{P}m$, is not necessarily an integer. As a result, the original information of $\hat{X}[k]$ is never repeated in any of the \hat{Y} indices. In fact, \hat{Y} would closely relate to an interpolated version of \hat{X} with the Dirichlet kernel.

Lemma 5.2 *Consider an OFDM symbol with N frequency subcarriers, indexed as $\{f_{-\frac{N}{2}}, \dots, 0, \dots, f_{\frac{N}{2}-1}\}$ with inter-frequency spacing of Δf , and a narrowband receiver that subsamples by $P \times$. If P^2 divides N , then the ideal filter parameters that meet all three requirements are: (1) $f_M^0 = f_{-\frac{N}{2}}$, (2) $(\frac{N}{P^2} - 1) \times \Delta f < \Delta S < \frac{N}{P^2} \times \Delta f$, and (3) $\Delta F = \frac{N}{P}(1 + \frac{1}{P}) \times \Delta f$.*

Proof of Lemma 5.2: First, we show that no two frequencies collide after aliasing. Let $q = \frac{N}{P}$, and assume that two frequencies f_α and f_β collide. Let f_α be k -th subcarrier (for $0 \leq k < P$) covered at the i -th passband ($0 \leq i < \lceil \frac{\Delta S}{\Delta f} \rceil$), and let f_β have k' and i' as corresponding indices. To collide after aliasing, $f_\alpha - f_\beta = (k - k')\Delta f + (i - i')\Delta f$ must be an integer multiple of $q\Delta f$. However, $|k - k'| \leq P - 1$ and $|i - i'| < \frac{N}{P^2}$. Thus $\frac{|f_\alpha - f_\beta|}{\Delta f} < (\frac{P-1}{P} + \frac{1}{P})q = q$, meaning we must have $f_\alpha - f_\beta = 0$, proving the first design requirement. Second, we note that P passbands that do not overlap (since $\Delta S < \Delta F$), and each passband covers exactly $\frac{N}{P^2}$ subcarriers. We therefore have

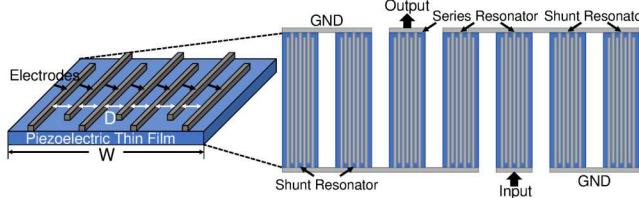


Figure 12: MEMS Spike-Train Filter Architecture

a total of $P \times \frac{N}{P^2} = q$ subcarriers that, as we just showed, do not overlap after aliasing. Therefore, after aliasing, each of the q subcarriers is covered exactly once, ensuring the second design requirement. Finally, we note that the smallest bin index is covered by the filter is $\min f_M = \frac{-N}{2}$, and the largest bin index is the last bin of the last passband, whose index can be computed as follows:

$$\begin{aligned}\max f_M &= \frac{-N}{2} + (P-1) \times \Delta F + \left\lceil \frac{\Delta S}{\Delta f} \right\rceil - 1 \\ &= \frac{-N}{2} + (P-1) \times \frac{N}{P} \left(1 + \frac{1}{P}\right) + \left(\frac{N}{P^2}\right) - 1 \\ &= -\frac{N}{2} + N - 1 = \frac{N}{2} - 1.\end{aligned}$$

Thus, the entire bandwidth (including $f_{-\frac{N}{2}}$ and $f_{\frac{N}{2}-1}$) is covered, ensuring the last design requirement.

B MEMS Spike-Train Filter

Spike-Train Filter Implementation: Following Lemma 5.2, we can derive the desired frequency response of the spike-train filter, and design MEMS resonators topology accordingly. For example, in our experiment, we used a 100 MHz 5G-like OFDM waveform with $N=2048$ subcarriers and a subcarrier spacing $\Delta f = 49$ kHz, and we down-sample the filtered waveform by a factor of $P=16$. According to Lemma 5.2, the desired filter should 16 spikes with a spike spacing of 6.64 MHz spanning the 100 MHz bandwidth, and each spike should have a width around 400 kHz.

We can design a spike-train filter leveraging the periodic resonance frequencies of a type of MEMS acoustic resonators that is commonly referred to as a LOBAR (Lateral Overtone Bulk Acoustic Resonator). As shown in Fig. 12, the LOBAR resonator consists of 12 electrodes on the top of a thin film made of the piezoelectric material $LiNbO_3$. And we combine seven resonators in a ladder filter topology [20] to build a filter circuit. As a result, the LOBAR resonator architecture

determines the spike frequencies, whereas the slight difference between different resonators determines the width of the spikes. For simplicity, here we only focus on these two key parameters of the spike-train filter response, since they are restricted by our channel recovery algorithm as described in Sec. 5. More details on the MEMS spike-train filter design can be found in [23].

(1) *The width of the film:* the spacing between spikes Δf is determined by the width of the thin film W as $\Delta f = v/W$, where v is the acoustic velocity in the piezoelectric material, which is ~ 4 km/s in our design. Therefore, to achieve the 6.6 MHz spike spacing, we design the film width W to be $\sim 660 \mu m$.

(2) *The film width difference between different shunt and series resonators:* the spike width ΔF of the spike-train filter equals to the resonant frequency difference between shunt and series resonators in the ladder filter, which is determined by the difference ΔW between shunt and series resonators: $\Delta F = fc \frac{\Delta W}{W}$. We design with piezoelectric film width to be 660 μm for series resonators and 660.26 μm for shunt resonators, which leads to $\Delta W = 0.26 \mu m$, so that the widths of the spikes are around 400 kHz.

C Updated Objective Function to Account for Residual CFO

ISLA captures the narrowband channel and wideband channel from different subframes. Thus, there is going to be an additional phase accumulation between the two measurements due to residual CFO. To address this, we slightly modify Eq.6 where we split the objective function into two separate L-2 norm minimizations, with the first term containing only the wideband channel h'_M , and the second term containing only the narrowband channel h'_{NB} . This objective function is given below:

$$\begin{aligned}\{\tau_l^*\}_{l=1}^L &= \arg \min_{\tau_1, \dots, \tau_L} \left(\|h'_M - V_M F_N \Psi (V_M F_N \Psi)^\dagger h'_M\|^2 \right. \\ &\quad \left. + \|h'_{NB} - V_{NB} F_N \Psi (V_{NB} F_N \Psi)^\dagger h'_{NB}\|^2 \right)\end{aligned}\quad (7)$$

s.t. $\tau_l \geq 0 \quad \forall l \in \{1, 2, \dots, L\}$

The modified objective function is now invariant to phase offsets between the two channels, and *ISLA* can solve this updated optimization using the same technique described in Sec. 6.

Accelerating Collective Communication in Data Parallel Training across Deep Learning Frameworks

Joshua Romero¹, Junqi Yin², Nouamane Laanait^{2*}, Bing Xie², M. Todd Young², Sean Treichler¹, Vitalii Starchenko², Albina Borisevich², Alex Sergeev^{3†}, Michael Matheson²
¹NVIDIA, Inc. ²Oak Ridge National Laboratory ³Carbon Robotics

Abstract

This work develops new techniques within Horovod, a generic communication library supporting data parallel training across deep learning frameworks. In particular, we improve the Horovod control plane by implementing a new coordination scheme that takes advantage of the characteristics of the typical data parallel training paradigm, namely the repeated execution of collectives on the gradients of a fixed set of tensors. Using a caching strategy, we execute Horovod’s existing coordinator-worker logic only once during a typical training run, replacing it with a more efficient decentralized orchestration strategy using the cached data and a global intersection of a bitvector for the remaining training duration. Next, we introduce a feature for end users to explicitly group collective operations, enabling finer grained control over the communication buffer sizes. To evaluate our proposed strategies, we conduct experiments on a world-class supercomputer — Summit. We compare our proposals to Horovod’s original design and observe 2× performance improvement at a scale of 6000 GPUs; we also compare them against `tf.distribute` and `torch.DDP` and achieve 12% better and comparable performance, respectively, using up to 1536 GPUs; we compare our solution against BytePS in typical HPC settings and achieve about 20% better performance on a scale of 768 GPUs. Finally, we test our strategies on a scientific application (STEMDL) using up to 27,600 GPUs (the entire Summit) and show that we achieve a near-linear scaling of 0.93 with a sustained performance of 1.54 exaflops (with standard error +- 0.02) in FP16 precision.

1 Introduction

The recent successes of Deep Neural Networks (DNNs) have encouraged continued investment across industries and domain sciences. Ranging from the traditional AI (e.g., image processing, speech recognition), to pharmaceutical and

*Nouamane Laanait conducted this research when he was with Oak Ridge National Laboratory.

†Alex Sergeev conducted this research when he was with Uber, Inc.

biomedical sciences (e.g., drug discovery), and to fusion, combustion and nuclear energy (e.g., disruption predictor, nuclear power plant) [29–34], more and more applications are actively exploiting ever-larger DNNs for production use.

With the growing applications of ever-larger DNNs, data parallelism in DNN training faces unprecedented challenges when synchronizing gradients¹ throughout distributed training runs. Deep learning (DL) frameworks, such as PyTorch [5] and TensorFlow [7], can exploit data parallelism for DNN training. In such a training run, an application creates multiple replicas of a model and distributes the replicas among a group of accelerators (e.g., CPUs, GPUs, TPUs, etc). Each accelerator executes on a different portion of training data across a number of *iterations*; at each iteration, it performs forward/backward pass computations independently, but synchronizes gradients (typically via global averaging) among the accelerators before applying weight updates (§2.1). In particular, accelerators synchronize tensors (multi-dimensional arrays) of gradients for the same set of parameters to ensure a globally consistent state for the model replicas.

This work advances collective communication in data parallel training. We propose several enhancements to Horovod [3] [25], a generic communication library designed to be independent to the framework runtimes, enabling its use across numerous popular DL frameworks with the same underlying backend implementation. Our ideas were motivated by two observations on Horovod. First, we observed that Horovod’s core design is not scalable (see Figure 3) as it relies on a coordinator-worker control plane to orchestrate collective operations. At larger scales, this design choice leads to the single coordinator becoming overwhelmed and leaves the application runtime dominated by the orchestration process. Second, we found that Horovod’s buffering mechanism (*Tensor Fusion*) fails to reliably generate optimal buffer sizes for efficient network bandwidth utilization (§2.2).

¹Centralized training (also called synchronous training) synchronizes gradients among accelerators; decentralized training (asynchronous training) synchronizes parameters [13] [14] [21]. This work optimizes centralized training and discusses gradient synchronization accordingly.

To address these inefficiencies, we improve the control plane with a new coordination scheme that takes advantage of characteristics of a typical data parallel training paradigm, namely the repeated execution of collectives on a fixed set of gradients (§2.1). Using a caching strategy, we execute Horovod’s existing coordinator-worker logic only once during a training run, replacing it with a more efficient decentralized orchestration strategy using a globally intersected bitvector for the remaining training duration (§3.1). Moreover, we introduce a feature for end users to explicitly group collective operations within Horovod, enabling finer grained control over the communication buffer sizes used for reductions.

While the implementation details vary, most DL-based communication libraries use similar design principles to optimize the performance of gradient synchronization. First, these libraries will employ mechanisms to facilitate overlapping of gradient synchronization and backward pass. That is, rather than waiting for gradients of all parameters to be computed and then synchronizing them across accelerators altogether at once, gradients will be synchronized actively as they are computed during the backward pass. Second, rather than launching a synchronization operator (e.g., AllReduce) for each gradient individually, the libraries employ bucketing/packing/fusion strategies (e.g., torch.DDP [18], tf.distribute [6], Horovod) to aggregate the gradients of multiple parameters and execute AllReduce on larger communication buffers for improved bandwidth utilization.

The contributions described in this work are mainly enhancements specific to Horovod, overcoming inefficiencies in its framework-agnostic design and original coordinator-worker strategy. The framework native communication libraries, like tf.distribute and torch.DDP, are closely integrated within their respective frameworks with access to internal details. With access to these details, the implementation of well-organized and performant communication and similar advanced features like grouping are simpler in these libraries. While the implementation details in this paper are Horovod specific, the proposed grouping technique is generally applicable to any other collective communication libraries.

In particular, we summarize our contributions as follows:

1. We implement a light weight decentralized coordination strategy by utilizing a response cache to enable Horovod to reuse coordination-related information collected at application runtime, accelerating the orchestration process.
2. We enable grouping to provide end users with explicit controls over tensor fusion in Horovod.
3. Our developments are incorporated in Horovod and publicly available in Horovod v0.21.0.
4. We conduct experiments to evaluate our solution on a world-class supercomputer — Summit. The results show that: 1) our solution outperforms Horovod’s existing strategies across scales consistently. 2) Compared to the framework native communication libraries such like tf.distribute and torch.DDP, we achieve comparable and/or better performance across scales

consistently. Compared to a PS (parameter server)-based communication library BytePS [24], we achieve 20% better performance using up to 768 GPUs. 3) we further evaluate our solution on a scale up to 27,600 GPUs (the entire Summit) and show that we achieve near-linear scaling of 0.93 with a sustained performance of 1.54 exaflops (with standard error ± 0.02) in FP16 precision.

2 Background and Motivation

2.1 Data Parallelism in DNN Training

For data parallelism in distributed DNN training, a typical application run usually executes an iterative learning algorithm (e.g., SGD) among a number of GPUs; each GPU works on an identical replica and the same set of parameters of a DNN model. Here, a parameter is the bias or weight of a DNN layer; the value of a parameter or the value of a parameter’s gradient is a multi-dimensional array, referred to as a *tensor*. In the run, a training dataset is partitioned into one or more equal-sized *batches*; each batch is processed on a different GPU. After a run starts, the model replicas, parameters, and the data structures of tensors are all fixed and determined.

During an iteration, each GPU updates parameters of a model replica by the following computational procedure: 1. the forward pass to compute loss. 2. the backward pass to compute gradients of the parameters. 3. the optimization step to update the parameters. In order to ensure model replicas are updated identically and remain in a globally consistent state, the gradients between GPUs are synchronized via averaging before updating parameters; this is referred to as *centralized learning*. Decentralized learning [13] [14] [21] maintains local consistency based on communication graphs ² and synchronizes parameters. Moreover, for both centralized and decentralized learning, GPUs synchronize the same set of parameters/gradients across iterations. In this work, we focus on centralized learning and discuss collective communication in gradient synchronization/reduction.

Observation ①. For a DNN training run on a DL framework, the model replicas and parameters are all fixed. Across iterations in the run, GPUs repeatedly synchronize the same set of tensors for parameters/gradients.

2.2 Communication Libraries for Gradient Synchronization

2.2.1 Framework-native Libraries

For data parallel training, the key communication operations that occurs are AllReduce operations which average gradients among GPUs. Within an iteration, the framework processes

²In decentralized learning, GPUs are structured into a communication graph (e.g., ring or torus); each GPU only synchronizes among its local neighbors on the graph.

on GPUs each generate a set of gradients during the backward pass that must be globally reduced before being used to update the model parameters.

DL frameworks typically use dependency graphs to schedule compute operations, the use of which may result in non-deterministic ordering of operations. This is because in general, the order of operations through the compute graph that satisfies all dependencies is not unique. As a result, the order of operations executed can vary across framework processes within a single iteration, or even between iterations on a single process. This leads to problems in handling gradient communication between processes, as the operations generating gradients may occur in varied orders across processes. If each framework process naively executes a blocking AllReduce on the gradients in the order they are produced locally, mismatches may arise leading to either deadlock or data corruption. A communication library for DL must be able to manage these non-deterministic ordering issues to ensure that AllReduce operations are executed between processes in a globally consistent order.

The framework-native communication libraries (e.g., `tf.distribute` and `torch.DDP`) are designed to be closely integrated within the framework and have direct access to internal details, such as the model definition and expected set of gradients to be produced each iteration. Access to this information enables these libraries to directly discern the communication required during an iteration and more easily implement a performant communication schedule. For example, `torch.DDP` is a wrapper around a model in PyTorch, and utilizes the information contained in the model about gradients to determine how to schedule AllReduce operations during an iteration. While access to this information can simplify the implementation of these communication libraries, it ties their implementations strictly to the frameworks they were designed to support.

2.2.2 Framework-agnostic Libraries

In contrast to the framework-native communication libraries, a framework-agnostic library avoids any reliance on internal framework details and makes communication scheduling decisions based on information deduced during runtime. This design choice enables the library to operate across numerous frameworks, but the lack of access to internal information presents unique challenges. This section discusses the design of Horovod, a framework-agnostic communication library.

Horovod is a generic communication library developed to execute collective communication in data parallel training on GPUs, CPUs and TPUs, and with support for various DL frameworks. It serves as a high-level communication library that leaves network routing details (e.g., network reordering) handled by lower-level libraries, such as MPI, etc. Without loss of generality, this section discusses how Horovod integrates with MPI and TensorFlow on GPUs. Assuming this scenario, a distributed training run has N identical model repli-

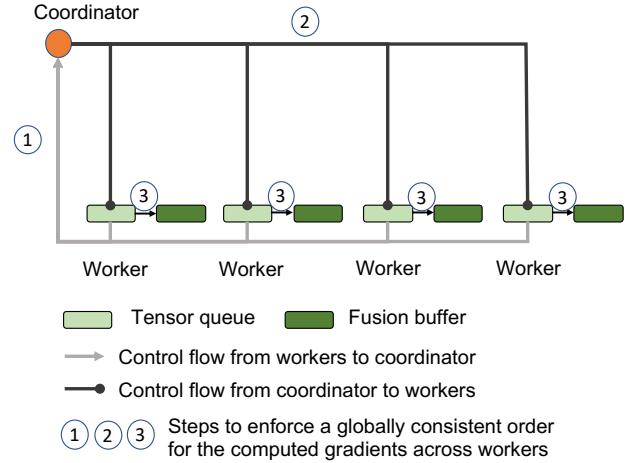


Figure 1: Coordinator-worker control model in Horovod’s original design. The coordination progresses in three steps (see details in §2.2.2): First, the coordinator gathers the lists of requests from all workers; Second, the coordinator processes the request lists, and then generates and broadcasts a response list when observing one or more common requests from all workers; Third, after receiving the response list, each worker proceeds to execute collective operations.

cas, and is executed on N GPUs managed by Horovod with MPI and TensorFlow. In the run, each GPU serves as both an MPI rank and a TensorFlow process³, which conducts computations for a model replica across iterations, with Horovod providing communication routines to synchronize gradients across TensorFlow processes.

This work introduces new techniques to Horovod after v0.15.2. In this section, we summarize the existing strategies based on v0.15.2. We use the terms *rank*, *process*, and GPU to refer to MPI rank, TensorFlow process, and their hosting GPU in turn, and use the terms *coordinator* and *worker* to refer to the Horovod threads spawned from the processes.

Similar to the framework-native libraries, Horovod must deal with the non-deterministic ordering of computations (discussed in §2.2.1). As it is agnostic to frameworks and lacking the knowledge of framework internal details, Horovod’s design uses a control plane to resolve the non-deterministic ordering issue, where a coordinator-worker communication model is adopted to orchestrate collective communication and ensure a globally consistent order of execution.

Figure 1 presents a simple diagram of Horovod’s control plane, with four threads each launched in a DL framework process. Particularly, the thread in Rank 0 serves as both the *coordinator* and a *worker*, and the other threads each serve as a different worker on a different GPU. During the course of a training run, the coordinator and workers execute the control logic periodically, with each execution referred

³For Horovod with TensorFlow, it is possible to use multi-GPUs per rank. But in production use, most users let each rank use a different GPU.

to as a *cycle*. In Horovod, the time between two sequential cycles is a configurable parameter with a default setting of 1 ms. To ensure synchronous cycles across Horovod threads, the communication operations in control plane (e.g., gather, broadcast) are blocking.

When a cycle starts, the coordinator first gathers lists of *requests* from all workers. Each request contains the metadata (e.g., tensor name, operation) that defines a specific collective operation on a specific tensor requested to be executed by the framework. The requests are collected from the worker’s local tensor queue and are structured as a *request list*.

Next, the coordinator processes the request lists and counts the submissions of each request (identified by tensor name) from workers. When the coordinator observes that a common request has been submitted by all workers, it prepares that request for execution by generating a corresponding *response*. The coordinator generates a list of responses and broadcasts the list to all workers. Here, each response contains the metadata (e.g., tensor names, data type, collective operation) that is used by the Horovod backend to execute a collective operation (e.g., AllReduce). Optionally, before broadcasting, the coordinator will preprocess the response list, aggregating multiple compatible responses into larger *fused* responses, a process referred to as *Tensor Fusion* in Horovod documentation.

After receiving the response list, each worker proceeds to execute collective operations, one operation per response in the received response list. The portion of the Horovod backend executing collective operations is referred to as the *data plane*. For each response, a worker will access required input tensor data from the framework, execute the requested collective operation, and populate the output tensors for the framework’s continued use. A key characteristic of this design is that the order of execution for collective operations is defined by the order of responses in the list produced by the coordinator. As such, a globally consistent ordering of collective operation execution is achieved across workers.

At a high-level, Horovod’s design can be described as a set of mailboxes, where each worker is free to submit request for collectives in any order to their assigned mailbox, and eventually retrieve the desired output. The control plane is responsible for coordinating these requests across mailboxes, ensuring that only requests submitted by all workers are executed and are executed in a globally consistent order. One observation from this analogy is that Horovod’s design is inherently unaware of any aspects of DL training, in particular that in typical DL workloads, a fixed set of gradient tensors will be repeatedly AllReduced during the course of a training run (discussed in §2.1). As a result, Horovod’s design unnecessarily communicates redundant information to the coordinator at every iteration, leading to poor scalability.

Beyond coordination alone, tensor fusion may cause inefficiency in the data plane. Ideally, the tensor fusion process will generate well balanced fused responses throughout training, yielding larger sized communication buffers for improved

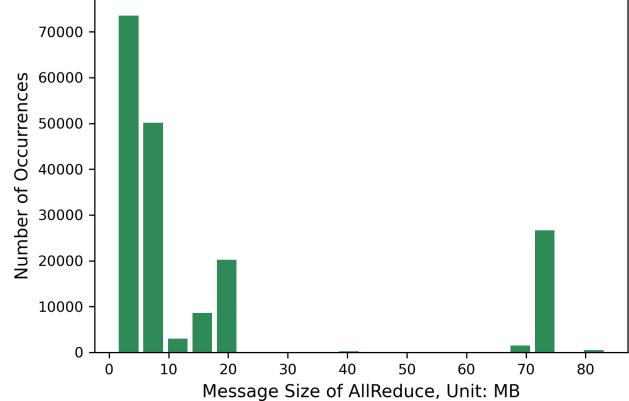


Figure 2: Histogram of AllReduce message size in Horovod’s original design of Tensor Fusion. We present the results of a training run of ResNet50 with 96 GPUs on Summit (§4.1).

network utilization. In practice, as the tensor fusion is closely tied to cycle that runs at an arbitrary user-defined tic rate, the resulting communication buffer sizes can be highly dynamic and varied, even when comparing iteration to iteration in a run. Figure 2 presents the fused AllReduce message sizes on ResNet50 as an example to illustrate the performance of tensor fusion. In summary, it is possible to have the Horovod cycles occur at favorable times during the training iteration, where the collective responses are well distributed across the Horovod cycles running during the iteration, resulting in correspondingly well-balanced fused communication message sizes. On the other hand, the Horovod cycles can occur at unfavorable times during the iteration, with some cycles completing with a few or even just one available collective response, yielding less efficient communication on smaller buffers. In the worst case, a single trailing gradient tensor for the iteration can be missed by all previous cycles run during the iteration, inducing additional latency equal to the user-defined cycle time, just to reduce a single gradient tensor.

We report the detailed information about the original design of Horovod’s control plane in the supplementary materials (Section 1), including pseudo code listings for Horovod coordinator-worker coordination logic and Horovod cycle, and the data structures for request list and response list.

Observation ②. The dynamic nature of tensor fusion can fail to generate buffer sizes for efficient network utilization. Thus, we are motivated to introduce a more explicit and strict control mechanism for tensor fusion that can improve performance.

2.2.3 Hierarchical Approach in Horovod

Kurth et al. [15] were the first to observe the scaling issue in Horovod’s control plane. In particular, the coordinator-worker coordination strategy was found to be highly inefficient. When increasing the number of workers, the time cost of the communication and processing grows linearly since the coordinator needs to communicate/process the request list

from each worker. Especially at large scale, the cost of this coordination strategy was found to quickly dominate the training runtime. Their proposed solution was to introduce a hierarchical tree-based variant of the original coordinator-worker control model, using a hierarchy of coordinators splitting up the coordination tasks. It is clear that this hierarchical control strategy outperforms the original control plane with a logarithmic complexity, but at the same time, it suffers from the same issue as the original strategy does: the hierarchical coordination strategy redundantly communicates metadata for repeated operations across iterations in a training run.

Beyond the hierarchical coordination strategy, the authors also introduced a hybrid/hierarchical AllReduce in Horovod’s data plane. Even with these improvements, their approach was not able to achieve efficient scaling with Horovod, requiring the introduction of a *gradient lag*. With gradient lag enabled, the gradients of a previous iteration are used to update weights in the current step, providing a longer window for overlapping the slower communication at scale with computation.

We present the hierarchical control plane in detail in the supplemental materials (Section 1) and discuss the performance of the hierarchical approach in Section 2.3.

Observation (3). Although existing Horovod solutions adopt different coordination strategies, they both fail to take advantage of characteristics of DL workloads and repeat the same metadata communications in the control plane across iterations in a training run.

2.3 Discussions on Horovod Performance

We focus on understanding the performance of existing Horovod solutions, including Horovod_MPI, Horovod_NCCL, and the hierarchical AllReduce (Hierarchical_AllReduce). Here, Horovod_MPI refers to the Horovod implementation with MPI for both the coordinator-worker communication in the control plane and AllReduce in the data plane. Horovod_NCCL refers to the implementation that uses MPI for control plane communication and NCCL for AllReduce in the data plane. In particular, NCCL v2.4.0 was used in this experiment, with tree-based communication algorithm options available along with existing systolic ring algorithm. Hierarchical_AllReduce represents the solution using MPI for the control plane communication and MPI+NCCL for the AllReduce in the data plane. In all three solutions, the coordinator-worker communication uses the control plane as shown in Figure 1. Moreover, all these solutions are available in Horovod [3].

We conducted experiments on STEMML (See supplementary materials Section 3), a scientific application developed to solve a long-standing inverse problem on scanning transmission electron micro-scoptic (STEM) data by employing deep learning. The DNN model in STEMML is a fully-convolutional dense neural network with 220 million parameters; each GPU generates/reduces 880MB of gradients at an

iteration. We ran the experiments on Summit supercomputer (§4.1), where each Summit node contains 6 GPUs.

We first consider the scalability results, shown in the left subfigure of Figure 3. It is clear that, after introducing the tree-based communication algorithms, Horovod_NCCL is able to deliver the best performance for all scales. When we increase the number of GPUs, Horovod_NCCL expands its lead in system throughput. For example, when using 6000 GPUs, it outperforms Hierarchical_AllReduce and Horovod_MPI by 3.2× and 5.4×, respectively.

Figure 3 (right subfigure) also reports the GPU utilization of the Horovod solutions across scales. The results show that, across all tested configurations, the GPU utilization is below 55%. When increasing the number of GPUs, the GPU utilization decreases progressively. We observed a much lower GPU utilization with 6000 GPUs (see Figure 6). This indicates that, although the NCCL-based AllReduce delivers good performance, the entire gradient reduction procedure in Horovod (e.g., coordination and execution) is highly inefficient. It leaves GPU resources underutilized and compromises system throughput. In this work, we argue that the inefficiency originates from both the control plane and AllReduce and introduce techniques (discussed in §3) to overcome these issues.

We limit the evaluation on Horovod_MPI to 1536 GPUs as we see noticeably poor performance. We skip the evaluations of the hierarchical tree-based coordinator-worker communication (Figure 1 in supplementary materials) and the gradient lag proposed in the hierarchical approach (§2.2.3), as they are currently neither included as part of Horovod nor publicly available. To summarize, Kurth et al. reported in [15] that, the entire hierarchical approach obtained the parallel efficiencies of ∼60% when using fully synchronous gradient reduction, only achieving above 90% on the Summit supercomputer with gradient lag enabled. In particular, researchers showed that gradient lag sometimes yields low training accuracy, and concluded that, without carefully tuning the related hyperparameters, this type of techniques is not generally applicable to DNN training [9, 10, 20]. Moreover, we show that our solution obtains up to 93% of parallel efficiency on Summit using a fully synchronous gradient reduction (discussed in §4.4), 1.5× better than the performance of the hierarchical approach without gradient lag reported in [15].

3 Boosting Collective Communication in DNN Training with Caching and Grouping

This work proposes to advance collective communication in centralized learning across various DL frameworks. We introduce new techniques to Horovod to improve its scalability and efficiency in both the control plane and the data plane. For the control plane, we develop a strategy to record the coordination information on the repeated requests for the same collective operations on the same parameters across

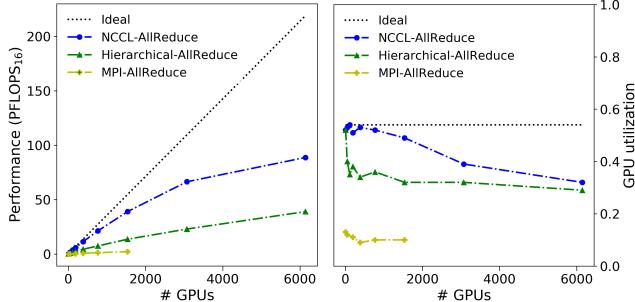


Figure 3: Performance and GPU utilization of existing Horovod strategies on STEMML workload.

iterations in a training run (discussed in §2.1). In particular, we develop a light weight decentralized coordination strategy by utilizing a *response cache*. This cache introduces a means for Horovod to store the metadata about the repeated collective requests at each worker locally and bypass the redundant coordinator-worker communication entirely after the cache is populated. Moreover, we introduce *grouping* as a feature to Horovod’s data plane. With grouping enabled, a user can request grouped collective operations for specific tensor groups, enforcing explicit control over Horovod’s tensor fusion. We later show in experiments (§4) that, these two techniques can lead to significant performance improvement and obtain near-linear scaling in the production runs on a world-class supercomputer. Our techniques are adopted by Horovod and are publicly accessible in v.0.21.0.

In general, our proposals are built within Horovod’s existing control logic (discussed in §2.2.2): we execute cycles to coordinate collective communication in DNN training; in our system, blocking communications are used to ensure synchronous cycles across workers and the network routing details (e.g., network reordering) are managed by lower-level communication libraries, such as MPI. Additionally, our modifications support both MPI and Gloo [2] libraries for control plane communication. We discuss the performance evaluation using MPI for control plane communication and either MPI or NCCL for data plane communication in Section 4.

3.1 Orchestrating Collective Communication with Caching

In contrast to the framework-native communication libraries like `tf.distribute` or `torch.DDP`, Horovod is designed to be generic. It utilizes lightweight bindings into frameworks to allow the Horovod runtime to process gradient reduction, and has no access to any data associated with the framework runtimes (e.g., iteration, parameters, models, etc.). In particular, Horovod interacts with DL frameworks via custom framework operations that enable the frameworks to pass a tensor and requested collective operation to the Horovod backend, and receive the output tensor after the collective is executed. These

custom operations are defined for each supported framework, as the mechanisms to share tensor data can vary between frameworks, but otherwise the remainder of the code base is generic. This design choice enables Horovod to work across numerous DL frameworks using the same underlying code, but at the same time, this generic design leads to the inefficiency at scale with its centralized coordinator-worker control plane.

As is summarized in **Observation 1**, in a typical data parallel training run, there is a fixed set of gradients that needs to be AllReduced across iterations. Horovod’s existing coordinator-worker design does not take advantage of this aspect of the workload, and will redundantly process the same collective communication requests through the coordinator at each iteration (**Observation 3**). Although this design choice allows Horovod to be dynamic and service any collective request submitted from workers, it is unnecessarily inefficient for the typical use case with a fixed set of repeated collective operations.

This section introduces a caching strategy that enables Horovod to capture and register repeated collective operations at runtime. With the cached metadata, we build a decentralized coordination among workers, replacing the existing strategy with significant performance improvement.

3.1.1 Response Cache

As Horovod does not have direct access to the framework-runtime metadata (e.g., iteration, tensors), any pattern of collective operations launched during a training run must be deduced at runtime based on prior collective requests observed. In order to capture the metadata about repeated collective operations, we introduce a *response cache* to Horovod. This cache can be used to identify repeated operations, as well as store associated *response* data structures generated by the coordinator to be reused without a repeated processing through the coordinator-worker process.

Each worker maintains a response cache locally. To construct the cache, Horovod threads will use the existing coordinator-worker control plane implementation. Specifically, workers send requests to the coordinator and receive a list of responses from the coordinator to execute. Instead of executing the collective operations immediately and destroying the response objects, the workers first store the response objects in a local cache, where each unique response is added to a linked-list structure. Additional tables are kept mapping tensor names to response objects in the cache as well as integer position indices in the linked list. A key characteristic of the cache design is that its structure is fully deterministic based on the order that response entries are added to the cache. In this design, the cache is populated using the list of responses received by the coordinator when a collective request is first processed. As the coordinator design already enforces a global ordering of responses, responses are added

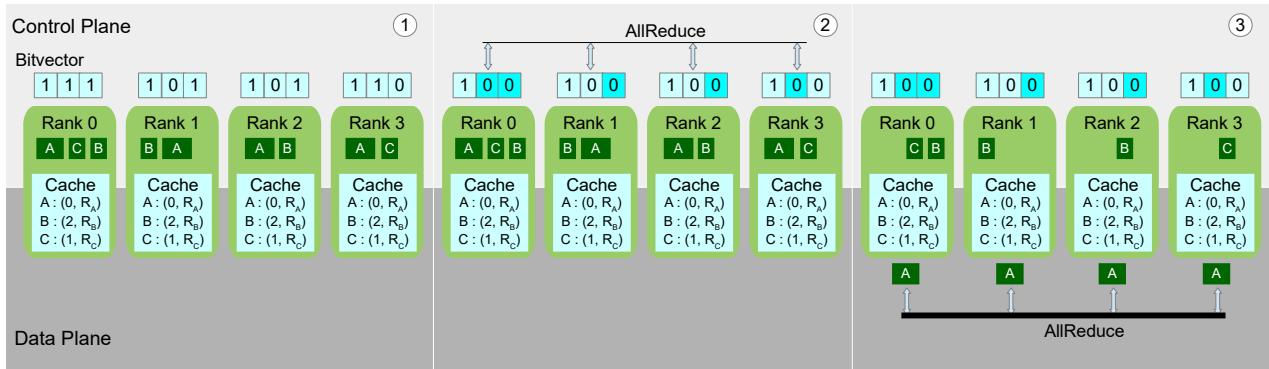


Figure 4: Illustration of AllReduce with Caching. We depict an example with 4 workers (0, 1, 2, 3) reducing 3 tensors (A, B, C). The strategy works in three steps: 1. Each worker populates a *bitvector*, setting bits according to entries in the *response cache* and the pending tensors in their local queues. 2. Workers synchronize the bitvectors via a global set intersection to identify common pending tensors. In this example, the bit associated with tensor A is shown as common across the workers. 3. Tensor A is sent to the data plane for AllReduce. When the AllReduce operation is done, Tensor A is removed from the queues on all workers.

to the cache on each worker in a globally consistent order which in turn ensures caches on each worker remain identical across workers. The data structure for each entry in the cache is the same as a response list discussed in Section 2.1. The cache implemented has a user-configurable capacity, with a default size of 1024 unique responses.

Using a combination of the cached responses and the globally consistent structure of the caches, a lightweight decentralized coordination scheme is enabled, as illustrated in Figure 4.

3.1.2 Cache-based Coordination with Response Cache and Bitvector

Once the response cache is created, it is utilized together with a bitvector to implement a lightweight decentralized coordination scheme. To achieve this, we take advantage of the fact that the response cache is constructed in a way that guarantees global consistency across workers. As a result, the structure of the response cache, in particular the index position of cached response entries, can be used to maintain a global indexing scheme of requests that are repeated that can be leveraged for coordination. We present the strategy in Figure 4, report the corresponding pseudo code in Algorithms 1 and 2, and summarize its procedure below.

1. At the start of a cycle, each worker performs the same operations as it does in the original design: it retrieves the pending requests from its local tensor queue, yielding a *RequestList*.
2. Each request in *RequestList* is checked against the response cache. If the request has an associated entry in the cache, the position of the cached entry is added to a set, *CacheBits*. Otherwise, this request does not have an associated cached entry and a flag is set to indicate that an uncached (i.e. previously unobserved) request is pending.

Algorithm 1 Horovod cycle with caching

```

1: procedure RUNCYCLEONCE
2:   RequestList  $\leftarrow$  PopMessagesFromQueue()
3:   CacheBitsg, UncachedInQueueg  $\leftarrow$  CacheCoordination(RequestList)
4:   UncachedRequestList  $\leftarrow$  []
5:   for M in RequestList do
6:     cached  $\leftarrow$  ResponseCache.cached(M)
7:     if cached then
8:       bit  $\leftarrow$  ResponseCache.GetCacheBit(M)
9:       if bit  $\notin$  CacheBitsg then
10:        PushMessageToQueue(M)  $\triangleright$  Replace messages corresponding to uncommon bit positions to framework queue for next cycle
11:      end if
12:    else
13:      UncachedRequestList.append(M)  $\triangleright$  Collect any uncached messages
14:    end if
15:   end for
16:   ResponseList  $\leftarrow$  ResponseCache.GetResponses(CacheBitsg)  $\triangleright$  Retrieve cached responses corresponding to common bit positions
17:   if not UncachedInQueueg then  $\triangleright$  All messages cached, skip master-worker coordination phase
18:     FusedResponseList  $\leftarrow$  FuseResponses(ResponseList)  $\triangleright$  Tensor Fusion
19:   else  $\triangleright$  Use master-worker coordination to handle uncached messages
20:     FusedResponseList  $\leftarrow$  MasterWorkerCoordination(UncachedRequestList, ResponseList)
21:   end if
22:   for R in FusedReponseList do
23:     ResponseCache.put(R)  $\triangleright$  Add response to cache
24:     PerformOperation(R)  $\triangleright$  Perform collective operation
25:   end for
26: end procedure

```

3. Each worker populates a bit vector, *BitVector*, setting bits corresponding to values in *CacheBits*. It also sets a bit to indicate whether it has uncached requests in its queue. The bit vectors across workers are globally intersected using an

Algorithm 2 Decentralized coordination with response cache and bitvector

```

1: procedure CACHECOORDINATION(RequestList)
2:   CacheBits  $\leftarrow \{\}$ , UncachedInQueue  $\leftarrow \text{False}$ 

3:   for M in RequestList do                                 $\triangleright$  Check for cached messages
4:     cached  $\leftarrow \text{ResponseCache.cached}(M)$ 
5:     if cached then
6:       bit  $\leftarrow \text{ResponseCache.GetCacheBit}(M)$ 
7:       CacheBits.insert(bit)                                 $\triangleright$  Collect bit positions for
                                                               cached entries
8:     else
9:       UncachedInQueue  $\leftarrow \text{True}$                        $\triangleright$  Record uncached message
                                                               exists
10:    end if
11:   end for

12:   BitVector  $\leftarrow \text{SetBitvector}(\text{CacheBits}, \text{UncachedInQueue})$   $\triangleright$  Set bits in local
      bitvector
13:   BitVectorg  $\leftarrow \text{Intersect}(\text{BitVector})$             $\triangleright$  AllReduce using binary
                                                               AND op to get global
                                                               bitvector
14:   CacheBitsg, UncachedInQueueg  $\leftarrow \text{DecodeBitVector}(\text{BitVector}_g)$   $\triangleright$  Get
      common bit positions and flag
15:   return CacheBitsg, UncachedInQueueg
16: end procedure
  
```

AllReduce with the binary AND operation, resulting in a globally reduced bitvector, BitVector_g . Through this operation, only bits corresponding to requests that are pending on all workers remain set, while others are zero.

4. Each worker decodes BitVector_g , collecting indices of any remaining set bits to form CacheBits_g , the set of cache indices corresponding to requests currently pending on all workers. Additionally, it extracts whether any worker has pending uncached requests in queue.
5. Each request in *RequestList* is checked against the entries in CacheBits_g . If the request has an associated cache entry but has a position not in CacheBits_g , this means that only a subset of workers have this cached request pending. This request is pushed back into the internal tensor queue to be checked again on a subsequent cycle. If the request has an associated cache entry with a position in CacheBits_g , this means that the request is pending on all workers and is ready for communication. The associated response is retrieved from the cache and added to the *ResponseList*. If the request is not cached, it is added to a list of uncached requests that needs to be handled via the coordinator-worker process.
6. If there are no uncached requests pending on any worker, the coordinator-worker process is completely skipped and workers proceed to process locally generated *ResponseLists* composed of response entries from the cache. Otherwise, uncached requests are handled via the coordinator-worker process, with the coordinator rank generating a *ResponseList* containing the cached response entries along with new responses corresponding to the uncached requests.

It is worth highlighting that with this cache-based control, the coordinator-worker logic is only executed during cycles where previously unobserved requests are submitted to Horovod. In cycles where all requests are cached (i.e. repeated), the coordinator-worker control plane is never exe-

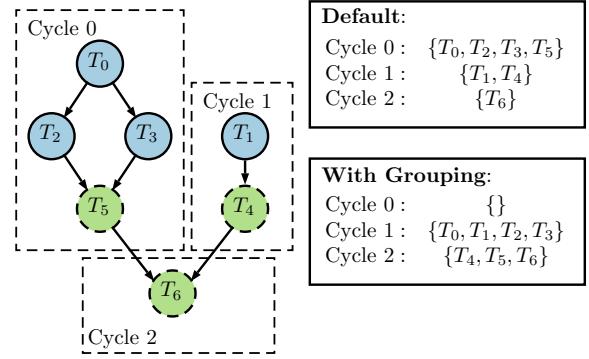


Figure 5: Illustration of Grouping. A task graph with nodes that generate requests T_n is depicted on the left, with the dashed boxes indicating requests visible to Horovod at 3 subsequent cycles. The nodes are colored to depict assignment to two groups (blue/solid borders and green/dashed borders). By default, a worker will submit all requests observed in a cycle to be processed/executed which can yield unbalanced sets of requests. With grouping enforced, requests are only submitted when complete groups are available.

cuted. For a typical DL workload with a fixed set of gradients to reduce every iteration, the response cache will eventually contain entries corresponding to this entire set. As a result, the poorly scaling coordinator-worker process will be skipped for all training iterations, except the first one, where all requests are initially observed and placed into the cache.

3.2 Grouping

The response cache described in the previous section addresses inefficiencies in the Horovod control plane. In this section, we describe a method to improve the data plane performance of Horovod through explicit grouping of AllReduce operations. In particular, we introduce a feature to Horovod that enables users to submit grouped collective operations, allowing explicit control over Horovod’s tensor fusion (§2.2.2).

As is shown in Figure 5, in place of submitting individual collective requests per tensor, a user can submit a grouped collective (e.g. `hvd.grouped_allreduce`) for multiple tensors. Collective requests submitted within a group are treated as a single request in Horovod’s control plane; that is, no request in the group is considered ready for the data plane until all requests in the group are submitted. As a result, the tensors within a group are guaranteed to be processed by the data plane during the same cycle and fused, along with any other responses ready for execution during the cycle.

This new grouping mechanism can be used to control how gradient AllReduces are scheduled during an iteration. In particular, the gradient AllReduce requests for a single iteration can be assigned to one or more groups to explicitly control the fused communication buffer sizes that Horovod

generates for gradient reduction, avoiding issues that can arise using the default dynamic fusing strategy as described in Section 2.2.2. To ease use, this functionality is exposed to users via a new argument, `num_groups` to Horovod’s high-level `DistributedOptimizer` wrapper. By setting this argument, the set of gradient tensors to be AllReduced within the iteration are evenly distributed into the number of groups specified. In the implementation described here, the gradients lists are split into groups of equal number of tensors, without consideration of buffer size.

Beyond this basic splitting, advanced users can achieve more optimal data plane communication performance by manually tuning the distribution of gradient tensors across the groups, to target fusion buffer sizes for improved network efficiency and/or achieving better overlap of communication and computation. We discuss the performance with different grouping configurations in Section 4.

We note that the framework native communication libraries like `torch.DDP` also support gradient fusion/bucketing and expose options to split gradient reduction into groups of approximately fixed message size. These native implementations generally leverage access to framework-level details, like information about the constructed model, to form these groups. As Horovod does not have access to these framework-level details directly, this grouping mechanism provides a means to provide such information via associating sets of tensors coming from the model to groups.

4 Experiment

4.1 Environment Setup

Hardware. We performed all experiments on Summit supercomputer [27] at the Oak Ridge Leadership Computing Facility. As the 2nd fastest supercomputer in the world, Summit is a 148.6 petaFLOPS (double precision) IBM-built supercomputer, consisting of 4,608 AC922 compute nodes with each node equipped with 2 IBM POWER9 CPUs and 6 NVIDIA V100 GPUs. Summit is considered as ideally suited for Deep Learning workloads, due to its node-local NVMe (called burst buffer) and Tensor Cores on V100 for faster low-precision operations. Moreover, its NVLink 2.0 and EDR InfiniBand interconnect provides 50 GB/s and 23 GB/s peak network bandwidths for intra-node and inter-node communication.

Software. The techniques proposed in this work are implemented based off Horovod v0.15.2 and have been incorporated in v0.21.0. We measured the performance with two communication backends, including NCCL v2.7.8 and Spectrum MPI (a variant of OpenMPI) v10.3.1.2. To evaluate the performance of our proposals across DL frameworks and to compare against the state-of-the-art communication libraries, we integrated our solutions in Horovod with TensorFlow (v2.3.1) and PyTorch (v1.6.0). We compared our solutions to `tf.distribute` in TensorFlow v2.4 (TensorFlow supports grouping since

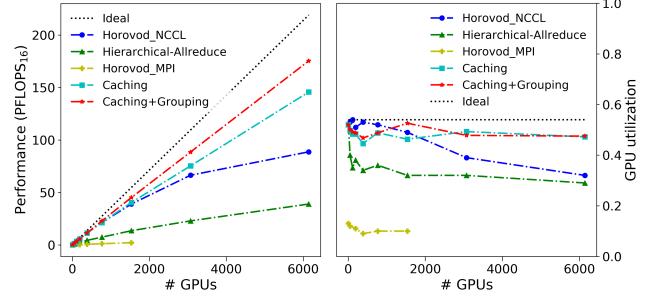


Figure 6: Performance and GPU utilization of Horovod’s strategies. We compare our new techniques to the existing Horovod implementations using STEMML (see Figure 3).

v2.4), `torch.DDP` in PyTorch v1.6.0, and BytePS (v0.2.5). In particular, BytePS is a deep learning framework that adopts PS (parameter server) as its communication model. BytePS is considered as an alternative to Horovod in a cloud environment. For `tf.distribute` and `torch.DDP`, we conducted the experiments with both NCCL and MPI; for BytePS, we conducted experiments simply with NCCL as BytePS does not support MPI. We configure BytePS in co-locate mode with one server and one worker per Summit node. We choose this configuration because it is recommended by the BytePS team as the best practice for high-performance computing (HPC) [1]. Moreover, we evaluated the scalability of our techniques with STEMML, where the results are from an earlier incarnation of this work based on Horovod v0.15.2 built with NCCL v2.4, but the conclusions are similar.

Workloads. We evaluated our solution on GPU-based workloads. Starting with the STEMML workload (message size 880MB per GPU), we compared our new techniques to the existing Horovod strategies (see Figures 3 and 6) with TensorFlow. We then broadened the experiments to compare with `tf.distribute`, `torch.DDP`, and BytePS on Resnet50 (102MB per GPU). Finally, we demonstrated our approach on ResNet50 and two more popular networks: EfficientNet-B0 (21MB per GPU) and VGG19 (574MB per GPU). We limit our interest in communication and use random synthetic data (of dimension (224, 224, 3)) as input to avoid impacts of I/O performance on the results. The training is in single precision with batch size of 64. We conducted the scalability experiments on the production code STEMML using TensorFlow. We briefly discuss STEMML in Section 2.3, report its source code in a GitHub repository (listed in Availability) and leave detailed documentation in Section 3 of the supplementary materials.

4.2 Evaluations on Horovod’s Strategies

This section evaluates the performance of various strategies in Horovod. We compare the performance of caching and grouping to the existing strategies across scales. Figure 6 reports the results, in which we follow the definitions about the

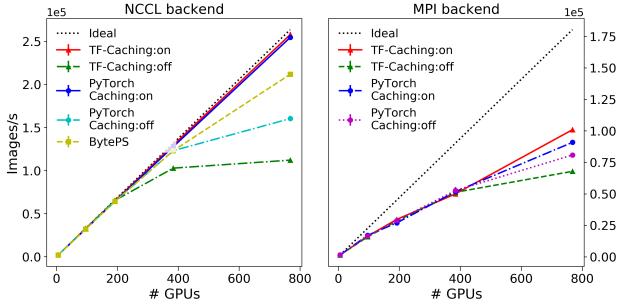


Figure 7: Performance of caching on ResNet50. We evaluate Horovod with caching enabled and disabled with both NCCL (left) and MPI (right) backends, and also compare the results to the performance of BytePS with NCCL (left).

existing strategies given in Section 2.3 and name the results of our techniques as *Caching* (cached-based coordination enabled) and *Caching+Grouping* (both caching and grouping enabled), respectively. Similar to Figure 3, we focus on analyzing performance (left subfigure) and GPU utilization (right subfigure). Here, performance refers to the floating-point operations performed per second (FLOPs). It is clear that our solutions outperform the existing strategies across scales consistently. When increasing the number of GPUs in use, the performance advantage grows rapidly. In particular, at the scale of 6000 GPUs, *Caching+Grouping* and *Caching* obtain $1.97\times$ and $1.64\times$ GPU performance improvement, and equally $1.48\times$ utilization improvement, over the Horovod baseline in NCCL-*AllReduce*. Accelerated by our techniques, 175 petaFLOPS in FP16 precision (more detailed discussion can be seen in supplementary materials Section 2) can be delivered with less than a quarter of Summit.

We conclude that our techniques achieve better performance than the existing strategies, especially at scale.

4.3 Evaluations across Frameworks and Communication Libraries

Next, we evaluate caching and grouping with both TensorFlow and Pytorch, and compare our techniques to `tf.distribute`, `torch.DDP`, and `BytePS`.

4.3.1 Caching and Grouping across Frameworks

We first analyze the caching performance on Horovod with TensorFlow and Pytorch. Figure 7 presents the results. It suggests that, for the results with both NCCL and MPI, the caching-enabled Horovod (`TF-Caching:on` and `PyTorch Caching:on`) first delivers equally good performance; and when increasing the number of GPUs to 384 and more, the caching-enabled Horovod delivers better performance consistently with both TensorFlow and Pytorch. In particular, compared to the caching-disabled Horovod (`TF-Caching:off` and `PyTorch Caching:off`) with NCCL on 768 GPUs, the

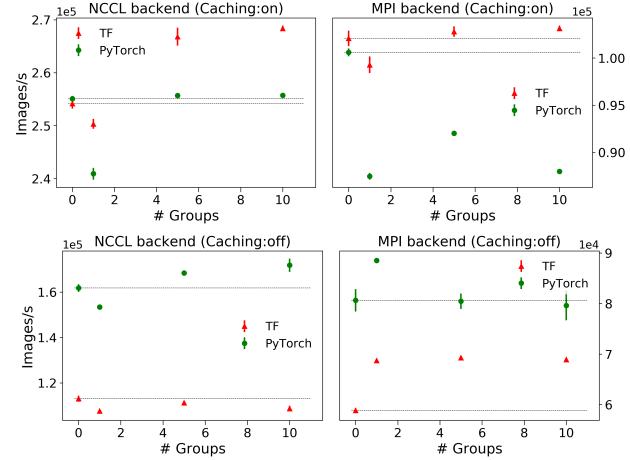


Figure 8: Performance of grouping on ResNet50. We evaluate Horovod with varied grouping configurations on 768 GPUs with caching enabled (top) and disabled (bottom) and with NCCL (left) and MPI (right) backends.

caching strategy achieves $2.5\times$ (`TF-Caching:on`) and $1.6\times$ (`PyTorch Caching:on`) performance improvement, respectively. Compared to the caching-disabled Horovod with MPI on 768 GPUs, the caching strategy achieves $1.53\times$ and $1.15\times$ performance improvement, respectively.

Figure 7 also presents the performance of BytePS (`BytePS`). It is shown clearly that BytePS delivers better performance than the cache-disabled Horovod consistently, and delivers equally good performance as the caching strategy does on the range of 6 GPUs — 384 GPUs, and delivers 20% lower performance than the caching strategy does on 768 GPUs. This suggests that, at larger scales, BytePS exhibits the scalability issue in typical HPC settings such as Summit. We leave the further study on the performance of BytePS on HPC clusters as future work.

Next, we report the grouping benefit in Figure 8. In the case with caching enabled (`Caching:on`), comparing to the case without grouping (# groups = 0), the training throughput on 768 GPUs with Horovod (NCCL backend) obtains a decent 5% boost with 5 or 10 tensor groups for TensorFlow, although the gain for PyTorch is less significant. For the much slower MPI backend, the improvement becomes marginal or negative. When the caching is turned off (`Caching:off`), there is a performance boost for PyTorch with the optimal group size, while for TensorFlow, it benefits mostly from grouping on the MPI backend. This indicates complicated interactions between the grouping and caching optimization.

To obtain a better understanding on the grouping behavior under different frameworks and communication fabrics, we plot the timing breakdown in Horovod for a 768-GPU training in Figure 9. For each iteration, the timing consists of two parts: coordination (control plane) and *AllReduce* (data plane). The timing for the *AllReduce* portion is further

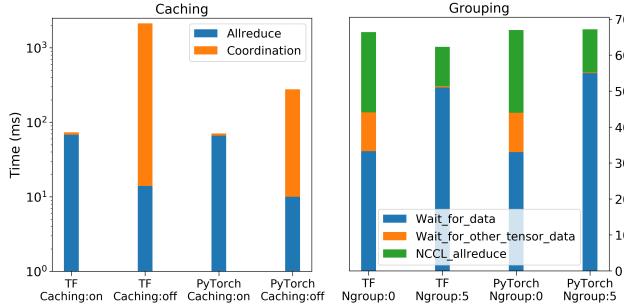


Figure 9: The inner timing breakdown in Horovod (NCCL backend) for a 768-GPU training with caching enabled and disabled (left) and grouping (# groups = 5) (right), respectively, during the training of ResNet50.

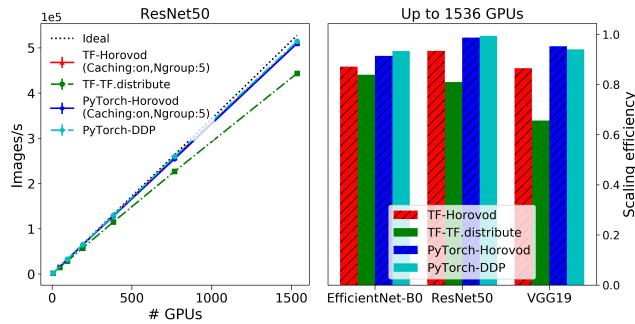


Figure 10: Scaling comparisons among Horovod, tf.distribute, and torch.DDP for the training of EfficientNet-B0, ResNet50, and VGG19. Training throughput (images/s) of ResNet50 (left). Scaling efficiency using up to 1536 GPUs (right).

split into wait (denoted [3] in Horovod as `WAIT_FOR_DATA` and `WAIT_FOR_OTHER_TENSOR_DATA` for time on waiting for framework to deliver gradient data and other data in the same fused collective, respectively) and actual communication (NCCL AllReduce). The case is slightly complicated for grouping. On one hand, the NCCL AllReduce time is almost cut in half because the grouped messages (orders of 10 MB) can better utilize network bandwidth; on the other hand, the wait time increases due to the coordination of groups. The overall performance of grouping depends on the trade-off between the aforementioned 2 factors. Too small number of groups (larger message and longer wait time) or too slow communication fabric (smaller or no gain in larger message communication) may result in worse performance with grouping, as indicated in Figure 8.

4.3.2 Evaluations across Communication Libraries

With both caching and grouping enabled, we compare the scaling efficiency of Horovod with tf.distribute and torch.DDP. To conduct a fair comparison, we ran all three libraries using a NCCL backend, and configure tf.distribute to use its

AllReduce mode (`MultiWorkerMirroredStrategy`), similar to Horovod and `torch.DDP`. In contrast to the experiments with TensorFlow v2.3.1 reported in the previous sections, this section contains experiments run using `tf.distribute` in TensorFlow v2.4 as it supports a comparable grouping feature and is a more recent release. Moreover, we disabled the `broadcast_buffers` option in `torch.DDP` to ensure that no additional collective operations outside the gradient AllReduces are performed during testing. We set the bucket size/pack size for grouping in `torch.DDP` and `tf.distribute` to 25MB as it is the default configuration for `torch.DDP`.

We present the results in Figure 10. As is shown clearly in the left subfigure, using up to 1536 GPUs, Horovod delivers 93% and 96% of scaling efficiencies with TensorFlow and PyTorch, respectively, while `tf.distribute` and DDP achieve 81% and 97% of the efficiencies, respectively. To further illustrate the scaling on different communication volumes, we plot the scaling efficiency for EfficientNet-B0, ResNet50, and VGG19 (right subfigure). Our approach shows an average of 12% better scaling than `tf.distribute` and a comparable performance to DDP, across model sizes, and the advantage becomes bigger as communication volume increases.

To obtain a better understanding of the performance of the three libraries, we profiled the training of ResNet50 with the libraries using Nsight Systems [4] (an NVIDIA profiling tool) and observed how well the AllReduce operations overlap with computation within a training iteration for each library. The results (see details in supplementary materials Section 4) show that all three libraries group tensors for AllReduce to a similar number of large buffers per iteration (4 or 5). In particular, we observed >95% of AllReduce overlapped with computation when using Horovod and `torch.DDP`, and the number dropped to ~75% when using `tf.distribute`.

We conclude that our solution performs well with both TensorFlow and PyTorch. Moreover, it delivers comparable and/or better performance than `tf.distribute` and `torch.DDP`, especially for large communication volumes.

4.4 Scaling Analysis on Production Code

This section evaluates the scaling efficiency of our solutions using a scientific DNN training code, STEMML. The purpose of the section is to demonstrate a use case that stresses the communication layer of DL training at extreme scales (e.g. 27k GPUs). Our expectation is that if a communication implementation can scale well in this scenario, it should be well suited to many other workloads operating with far fewer tasks. Beyond scaling efficiency, we also evaluate the power consumption and overall performance of the production runs of STEMML on the fully-scaled Summit, and leave the detailed documentation (e.g., the metrics and evaluations) to Section 2 in the supplementary materials, due to space limitations.

Figure 11 presents the scaling results. With both caching and grouping enabled, Horovod achieves a scaling efficiency

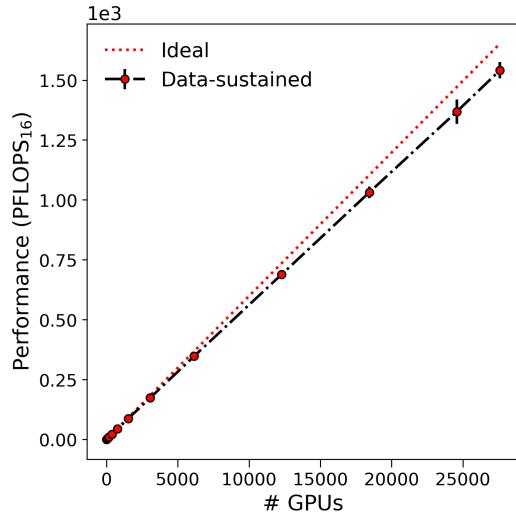


Figure 11: Scaling efficiency of STEMDL using up to 27,600 GPUs, the entire Summit.

of 0.93 at 27,600 GPUs and reach a sustained performance of 1.54 exaflops (with standard error ± 0.02) and a peak performance of 2.15 exaflops (with standard error ± 0.02) in FP16 precision. Moreover, on a single GPU, our proposals attain 59.67 and 83.92 teraflops as the sustained and peak performance, respectively. It suggests that each GPU achieves 49.7% and 70% of the theoretical peak performance of a V100 (120 teraflops) as its sustained and peak performance. To the best of our knowledge, it exceeds the single GPU performance of all other DNN trained on the same system to date.

We conclude that our techniques can attain near-linear scaling on up to 27,600 GPUs.

5 Related Work

Other than collective AllReduce, another popular scheme for data parallelism is parameter server. Incorporated with many acceleration techniques such as hierarchical strategy, priority-based scheduling, etc, BytePS [12, 24] has shown better scaling performance than Horovod in a cloud environment where parameter servers run on CPU-only nodes, because the network bandwidth can be more efficiently utilized⁴. We compared our solutions with BytePS on a typical HPC setting and the results (see Figure 7) show that our techniques perform better in such settings.

One promising direction is to further reduce the communication volume via compression [8, 11, 26, 35, 36], decentralized learning [13, 14, 19], or staled/asynchronized communication [9, 10, 20]. The compression techniques include quantization, sparsification, sketching, etc, and the combined

⁴In current ring-based AllReduce (as implemented in NCCL), each model replica sends and receives $2(N - 1)/N$ times gradients (N being number of GPUs), so the total message volume transferred in network per model is 2x of the gradient volume for large N .

method [22] has shown 2 orders of magnitude in communication volume reduction without loss of accuracy. For decentralized learning, depending on the communication graphs for model replicas, the communication complexity is reduced to $O(\text{Deg}(\text{graph}))$ independent of scale. Staled/asynchronized communication can boost the communication performance by relaxing the synchronization requirement across model replicas, which usually comes with some cost in model convergence. These developments are orthogonal to our approach, and in principle, our techniques can apply on top of them.

Beyond proposals for improving collective communication in DNN training. Kungfu [23] is proposed to auto-tune the parameters in both DNN models and DL frameworks based on runtime monitoring data. This effort is complementary to ours: we propose techniques in Horovod with introduction of parameters that may benefit tremendously from appropriate tuning. Another significant recent study [28] proposed Drizzle to improve large scale streaming systems with group scheduling and pre-scheduling shuffles. Similar to our approach, Drizzle reused scheduling decisions to reduce coordination overhead across micro-batches. But different to our decentralized coordination proposal, Drizzle amortized the overhead of centralized scheduling.

6 Conclusion

We have shown that by introducing a new coordination strategy and a grouping strategy we exceed the state of the art in scaling efficiency. This opens up, in particular, opportunities in exploiting the different levels of parallelism present in many systems (e.g. intra-node vs inter-node) such as Summit to train even larger DNN models.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Shivararam Venkataraman, for their invaluable comments that improved this paper. This research was partially funded by a Lab Directed Research and Development project at Oak Ridge National Laboratory, a U.S. Department of Energy facility managed by UT-Battelle, LLC. An award of computer time was provided by the INCITE program. This research also used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

Availability

The proposed techniques have been upstreamed to the Horovod main distribution [3]. The code for full Summit distributed training and the software for data generation are made public [16, 17].

References

- [1] BytePS Best Practice. <https://github.com/bytedance/bytdeps/blob/master/docs/best-practice.md>.
- [2] Gloo. <https://github.com/facebookincubator/gloo>.
- [3] Horovod. <https://github.com/horovod/horovod>.
- [4] Nvidia Nsight. <https://developer.nvidia.com/nsight-systems>.
- [5] PyTorch. <https://pytorch.org/>.
- [6] tf.distribute in TensorFlow. https://www.tensorflow.org/api_docs/python/tf/distribute.
- [7] 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 *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.
- [8] Naman Agarwal, Ananda Theertha Suresh, Felix Yu, Sanjiv Kumar, and H. Brendan McMahan. cpSGD: Communication-efficient and differentially-private distributed SGD. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*, 2018.
- [9] Suyog Gupta, Wei Zhang, and Fei Wang. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 2017.
- [10] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS'13)*, 2013.
- [11] Nikita Ivkin, Daniel Rothchild, Enayat Ullah, Vladimir Braverman, Ion Stoica, and Raman Arora. Communication-efficient distributed SGD with sketching. In *Advances in Neural Information Processing Systems (NIPS'19)*, 2019.
- [12] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.
- [13] Anastasia Koloskova*, Tao Lin*, Sebastian U Stich, and Martin Jaggi. Decentralized deep learning with arbitrary communication compression. In *Proceedings of the International Conference on Learning Representations (ICLR'20)*, 2020.
- [14] Anastasia Koloskova, Sebastian U Stich, and Martin Jaggi. Decentralized stochastic optimization and gossip algorithms with compressed communication. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, 2019.
- [15] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Maphesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*, 2018.
- [16] Nouamane Laanait, Michael A Matheson, Suhas Somnath, Junqi Yin, and USDOE. STEMDL. <https://www.osti.gov//servlets/purl/1630730>, 9 2019.
- [17] Nouamane Laanait, Junqi Yin, and USDOE. NAMSA. <https://www.osti.gov//servlets/purl/1631694>, 8 2019.
- [18] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch distributed: Experiences on accelerating data parallel training. *Very Large Data Bases Conference (VLDB'20)*, 2020.
- [19] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. Pipe-SGD: A decentralized pipelined SGD framework for distributed deep net training. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*, 2018.
- [20] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS'15)*, 2015.

- [21] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, 2018.
- [22] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *Proceedings of the International Conference on Learning Representations (ICLR'18)*, 2018.
- [23] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.
- [24] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, 2019.
- [25] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.
- [26] Ryan Spring, Anastasios Kyrillidis, Vijai Mohan, and Anshumali Shrivastava. Compressing gradient optimizers via count-sketches. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, 2019.
- [27] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenberg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. The design, deployment, and evaluation of the coral pre-exascale systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'18)*, 2018.
- [28] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, 2017.
- [29] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing output bottlenecks in a supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'12)*, 2012.
- [30] Bing Xie, Jeffrey Chase, David Dillow, Scott Klasky, Jay Lofstead, Sarp Oral, and Norbert Podhorszki. Output performance study on a production petascale filesystem. In *HPC I/O in the Data Center Workshop (HPC-IODC'17)*, 2017.
- [31] Bing Xie, Yezhou Huang, Jeffrey Chase, Jong Youl Choi, Scott Klasky, Jay Lofstead, and Sarp Oral. Predicting output performance of a petascale supercomputer. In *Proceedings of the International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'17)*, 2017.
- [32] Bing Xie, Sarp Oral, Christopher Zimmer, Jong Youl Choi, David Dillow, Scott Klasky, Jay Lofstead, Norbert Podhorszki, and Jeffrey S Chase. Characterizing output bottlenecks of a production supercomputer: Analysis and implications. *ACM Transactions on Storage (TOS'20)*, 2020.
- [33] Bing Xie, Zilong Tan, Phil Carns, Jeff Chase, Kevin Harms, Jay Lofstead, Sarp Oral, Sudharshan Vazhkudai, and Feiyi Wang. Applying machine learning to understand write performance of large-scale parallel filesystems. In *the 4TH International Parallel Data Systems Workshop (PDSW'19)*, 2019.
- [34] Bing Xie, Zilong Tan, Phil Carns, Jeff Chase, Kevin Harms, Jay Lofstead, Sarp Oral, Sudharshan S Vazhkudai, and Feiyi Wang. Interpreting write performance of supercomputer I/O systems with regression models. In *Proceedings of the 36th IEEE International Parallel and Distributed Processing Symposium (IPDPS'21)*, 2021.
- [35] Min Ye and Emmanuel Abbe. Communication-computation efficient gradient coding. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, 2018.
- [36] Yue Yu, Jiaxiang Wu, and Longbo Huang. Double quantization for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems (NIPS'19)*, 2019.

Cocktail: A Multidimensional Optimization for Model Serving in Cloud

Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma,
Mahmut Taylan Kandemir, Chita R. Das

The Pennsylvania State University, University Park, PA

{jashwant, cyan, prashanth, mtk2, cxd12}@psu.edu, bikash.nitrkl@acm.org

Abstract

With a growing demand for adopting ML models for a variety of application services, it is vital that the frameworks serving these models are capable of delivering highly accurate predictions with minimal latency along with reduced deployment costs in a public cloud environment. Despite high latency, prior works in this domain are crucially limited by the accuracy offered by individual models. Intuitively, model ensembling can address the accuracy gap by intelligently combining different models in parallel. However, selecting the appropriate models dynamically at runtime to meet the desired accuracy with low latency at minimal deployment cost is a nontrivial problem. Towards this, we propose *Cocktail*, a cost effective ensembling-based model serving framework. *Cocktail* comprises of two key components: (i) a dynamic model selection framework, which reduces the number of models in the ensemble, while satisfying the accuracy and latency requirements; (ii) an adaptive resource management (RM) framework that employs a distributed proactive autoscaling policy, to efficiently allocate resources for the models. The RM framework leverages transient virtual machine (VM) instances to reduce the deployment cost in a public cloud. A prototype implementation of *Cocktail* on the AWS EC2 platform and exhaustive evaluations using a variety of workloads demonstrate that *Cocktail* can reduce deployment cost by $1.45\times$, while providing $2\times$ reduction in latency and satisfying the target accuracy for up to 96% of the requests, when compared to state-of-the-art model-serving frameworks.

1 Introduction

Machine Learning (ML) has revolutionized user experience in various cloud-based application domains such as product recommendations [70], personalized advertisements [44], and computer vision [13, 43]. For instance, Facebook [44, 82] serves trillions of inference requests for user-interactive applications like ranking new-feeds, classifying photos, etc. It is imperative for these applications to deliver accurate predictions at sub-millisecond latencies [27, 34, 35, 39, 44, 83] as they critically impact the user experience. This trend is expected to perpetuate as a number of applications adopt a variety of ML models to augment their services. These ML models are typically trained and hosted on cloud platforms as service endpoints, also known as *model-serving* framework [6, 28, 60]. From the myriad of ML flavours, Deep Neural Networks

(DNNs) [54] due to their multi-faceted nature, and highly generalized and accurate learning patterns [45, 73] are dominating the landscape by making these model-serving frameworks accessible to developers. However, their high variance due to the fluctuations in training data along with compute and memory intensiveness [59, 65, 84] has been a major impediment in designing models with high accuracy and low latency. Prior model-serving frameworks like InFaas [83] are confined by the accuracy and latency offered by such individual models.

Unlike single-model inferences, more sophisticated techniques like *ensemble learning* [15] have been instrumental in allowing model-serving to further improve accuracy with multiple models. For example, by using the ensembling ¹ technique, images can be classified using multiple models *in parallel* and results can be combined to give a final prediction. This significantly boosts accuracy compared to single-models, and for this obvious advantage, frameworks like Clipper [27] leverage ensembling techniques. Nevertheless, with ensembling, the very high resource footprint due to sheer number of models that need to be run for each request [27, 56], exacerbates the public cloud deployment costs, as well as leads to high variation in latencies. Since cost plays a crucial role in application-provider consideration, it is quintessential to minimize the deployment costs, while maximizing accuracy with low latency. Hence, the non-trivial challenge here lies in making the cost of ensembling predictions analogous to single model predictions, while satisfying these requirements.

Studying the state-of-the-art ensemble model-serving frameworks, we observe the following critical shortcomings:

- Ensemble model selection policies used in frameworks like Clipper [27] are static, as they *ensemble all available models* and focus solely on minimizing loss in accuracy. This leads to higher latencies and further inflates the resource footprint, thereby accentuating the deployment costs.
- Existing ensemble weight estimation [87] has *high computational complexity* and in practice is limited to a small set of off-the-shelf models. This leads to significant loss in accuracy. Besides, employing linear ensembling techniques such as model averaging are compute intensive [80] and not scalable for a large number of available models.
- Ensemble systems [27, 80] are *not focused towards model deployment* in a public cloud infrastructure, where resource

¹We refer to ensemble-learning as ensembling throughout the paper.

selection and procurement play a pivotal role in minimizing the latency and deployment costs. Further, the resource provisioning strategies employed in single model-serving systems are *not directly extendable* to ensemble systems.

These shortcomings collectively motivate the central premise of this work: *how to solve the complex optimization problem of cost, accuracy and latency for an ensembling framework?* In this paper, we present and evaluate *Cocktail*², which to our knowledge is the first work that proposes a cost-effective model-serving system by exploiting ensembling techniques for classification-based inference, to deliver high accuracy and low latency predictions. *Cocktail* adopts a three-pronged approach to solve the optimization problem. First, it uses a dynamic model selection policy to significantly reduce the number of models used in an ensemble, while meeting the latency and accuracy requirements.

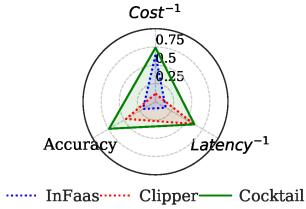


Figure 1: Benefits of *Cocktail*. Results are normalized (higher the better).

as they can be 70-90% cheaper [3] than traditional VMs. *Cocktail*, by coalescing these benefits, is capable of operating in a region of optimal cost, accuracy and latency (shown in Figure 1) that prior works cannot achieve. Towards this, the **key contributions** of the paper are summarized below:

1. By characterizing accuracy vs. latency of ensemble models, we identify that prudently selecting a subset of available models under a given latency can achieve the target accuracy. We leverage this in *Cocktail*, to design a novel dynamic model selection policy, which ensures accuracy with significantly reduced number of models.
2. Focusing on classification-based inferences, it is important to minimize the bias in predictions resulting from multiple models. In *Cocktail*, we employ a per-class weighted majority voting policy, that makes it scalable and effectively breaks ties when compared to traditional weighted averaging, thereby minimizing the accuracy loss.
3. We show that uniformly scaling resources for all models in the ensemble leads to over-provisioning of resources and towards minimizing it, we build a distributed weighted auto-scaling policy that utilizes the *importance sampling* technique to proactively allocate resources to every model. Further, *Cocktail* leverages transient VMs as they are cheaper, to drastically minimize the cost for hosting model-serving infrastructure in a public cloud.

²Cocktail is ascribed to having the perfect blend of models in an ensemble.

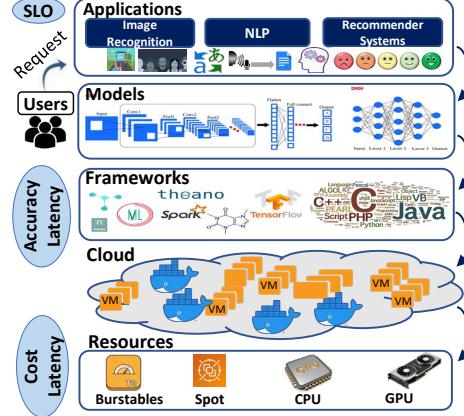


Figure 2: The overall framework for model-serving in public cloud.

4. We implement a prototype of *Cocktail* using both CPU and GPU instances on AWS EC2 [5] platform and extensively evaluate it using different request-arrival traces. Our results from exhaustive experimental analysis demonstrate that *Cocktail* can minimize deployment cost by $1.4\times$ while meeting the accuracy for up-to 96% of the requests and providing $2\times$ reduction in latency, when compared to state-of-the-art model serving systems.
5. We show that ensemble models are inherently fault-tolerant over single models, since in the former, failure of a model would incur some accuracy loss without complete failure of the requests. It is observed from our failure-resilience results that *Cocktail* can adapt to instance failures by limiting the accuracy loss within 0.6%.

2 Background and Motivation

We start by providing a brief overview of model-serving in public cloud and ensembling, followed by a detailed analysis of their performance to motivate the need for *Cocktail*.

2.1 Model Serving in Public Cloud

Figure 2 shows the overall architecture of a model-serving framework. There are diverse applications that are typically developed, trained and hosted as web services. These services allow end-users to submit queries via web server interface. Since these inference requests are often user-facing, it is imperative to administer them under a strict service level objective (SLO). We define SLO as the end-to-end response latency required by an application. Services like Ads and News Feed [39, 44] would require SLOs within 100ms, while facial tag recommendation [83] can tolerate up to 1000ms. A myriad of model architectures are available to train these applications which by themselves can be deployed on application frameworks like TensorFlow [1], PyTorch [62] etc. Table 1 shows the different models available for image prediction, that are pretrained on Keras using ImageNet [29] dataset. Each model has unique accuracy and latencies depending on the model architecture. Typically denser models are designed with more parameters (ex. NASLarge) to classify complex

Model (Acronym)	Params (10k)	Top-1 Accuracy(%)	Latency (ms)	P_f
MobileNetV1 (MNet)	4,253	70.40	43.45	10
MobileNetV2 (MNetV2)	4,253	71.30	41.5	10
NASNetMobile (NASMob)	5,326	74.40	78.18	3
DenseNet121 (DNet121)	8,062	75.00	102.35	3
DenseNet201 (DNet201)	20,242	77.30	152.21	2
Xception (Xcep)	22,910	79.00	119.2	4
Inception V3 (Incep)	23,851	77.90	89	5
ResNet50-V2 (RNet50)	25,613	76.00	89.5	6
Resnet50 (RNet50)	25,636	74.90	98.22	5
IncepResnetV2 (IRV2)	55,873	80.30	151.96	1
NasNetLarge (NasLarge)	343,000	82.00	311	1

Table 1: Collection of pretrained models used for image classification.

classes of images. These 11 models are a representative set to classify all images belonging to 1000 classes in Imagenet . Depending on the application type, the maximum ensemble size can vary from tens to hundreds of models.

The entire model framework is typically hosted on resources like VMs or containers in public cloud. These resources are available in different types including CPU/GPU instances, burstables and transient instances. Transient instances [69] are similar to traditional VMs but can be revoked at any time by the cloud provider with an interruption notice. The provisioning latency, instance permanence and packing factor of these resources have a direct impact on the latency and cost of hosting model-serving. We explain instance “packing factor” and its relationship with latency in Section 2.3.2. In this paper, we focus on improving the accuracy and latency from the model selection perspective and consider instances types from a cost perspective. A majority of the model serving systems [6, 83, 86] in public cloud support individual model selection from available models. For instance, InFaas [83] can choose variants among a same model to maintain accuracy and latency requirements. However, denser models tend to have up to $6\times$ the size and twice the latency of smaller models to achieve increased accuracy of about 2-3%. Besides using dense models, ensembling [15] techniques have been used to achieve higher accuracy.

Why Ensembling? An Ensemble is defined as a set of classifiers whose individual decisions combined in some way to classify new examples. This has proved to be more accurate than traditional single large models because it inherently reduces incorrect predictions due to variance and bias. The commonly used ensemble method in classification problems is bagging [33] that considers homogeneous weak learners, learns them independently from each other in parallel, and combines them following some kind of deterministic averaging process [18] or majority voting [49] process. For further details on ensemble models, we refer the reader to prior works [14, 57, 58, 61, 64, 77, 78, 88].

2.2 Related Work

Ensembling in practice: Ensembling is supported by commercial cloud providers like Azure ML-studio [11] and AWS Autogluon [31] to boost the accuracy compared to single models. Azure initially starts with 5 models and scales up to

Features	Clipper [27]	Rafiki [80]	Infaas [83]	MArk [86]	Sagemaker	Swayam [34]	Cocktail
Predictive Scaling	✗	✗	✗	✓	✗	✓	✓
SLO Guarantees	✓	✗	✓	✓	✗	✓	✓
Cost Effective	✗	✗	✓	✓	✗	✗	✓
Ensembling	✓	✓	✗	✗	✓	✗	✓
Heterogeneous Instances	✗	✓	✓	✓	✓	✗	✓
Dynamic ensemble selection	✗	✗	✗	✗	✗	✗	✓
Model abstraction	✓	✓	✓	✗	✗	✗	✓

Table 2: Comparing *Cocktail* with other related frameworks.

200 using a hill-climb policy [17] to meet the target accuracy. AWS combines about 6-12 models to give the best possible accuracy. Users also have the option to manually mention the ensemble size. Unlike them, *Cocktail*’s model selection policy tries to right-size the ensemble for a given latency, while maximizing accuracy.

Model-serving in Cloud: The most relevant prior works to *Cocktail* are InFaas [83] and Clipper [27], which have been extensively discussed and compared to in Section 6. Recently FrugalML [20] was proposed to cost-effectively choose from commercial MLaaS APIs. While striking a few similarities with *Cocktail*, it is practically limited to image-classification applications with very few classes and does not address resource provisioning challenges. Several works [37, 38] like MArk [86] proposed SLO and cost aware resource procurement policies for model-serving. Although our heterogeneous instance procurement policy has some similarities with MArk, it is significantly different because we consider ensemble models. Rafiki [80] considers small model sets and scales up and down the ensemble size by trading off accuracy to match throughput demands. However, *Cocktail*’s resource management is more adaptive to changing request loads and does not drop accuracy. Pretzel [52] and Inferline [26] are built on top of Clipper to optimize the prediction pipeline and cost due to load variations, respectively. Many prior works [2, 25, 35, 63, 74, 75] have extensively tried to reduce model latency by reducing overheads due to shared resources and hardware interference. We believe that our proposed policies can be complementary and beneficial to these prior works to reduce the cost and resource footprint of ensembling. There are mainstream commercial systems which automate single model-serving like TF-Serving [60], SageMaker [6], AzureML [10], Deep-Studio [28] etc.

Autoscaling in Public Cloud: There are several research works that optimize the resource provisioning cost in public cloud. These works are broadly categorized into: (i) multiplexing the different instance types (e.g., Spot, On-Demand) [12, 23, 34, 41, 42, 68, 79], (ii) proactive resource provisioning based on prediction policies [34, 36, 40, 41, 69, 86]. *Cocktail* uses similar load prediction models and auto-scales VMs in a distributed fashion with respect to model ensembling. Swayam [34] is relatively similar to our work as it han-

Baseline(BL)	NASLarge	IRV2	Xception	DNet121	NASMob
#Models	10	8	7	5	2
BL_Latency	311(ms)	152(ms)	120(ms)	100(ms)	98(ms)
E_Latency	152(ms)	120(ms)	103(ms)	89(ms)	44(ms)

Table 3: Comparing latency of Ensembling (E_Latency) with single (baseline) models.

dles container provisioning and load-balancing, specifically catered for single model inferences. *Cocktail*'s autoscaling policy strikes parallels with Swayam's distributed autoscaling; however, we further incorporate novel importance sampling techniques to reduce over-provisioning for under-used models. Table 2 provides a comprehensive comparison of *Cocktail* with the most relevant works across key dimensions.

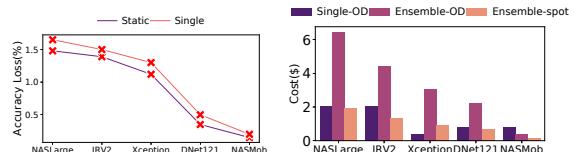
2.3 Pros and Cons of Model Ensembling

In this section, we quantitatively evaluate (i) how effective ensembles are in terms of accuracy and latency compared to single models, and (ii) the challenges in deploying ensemble frameworks in a cost-effective fashion on a public cloud. For relevance in comparison to prior work [27, 83] we chose image inference as our ensemble workload. While ensembling is applicable in other classification workloads like product recommendations [24, 53], text classification [71] etc, the observations drawn are generic and applicable to other applications.

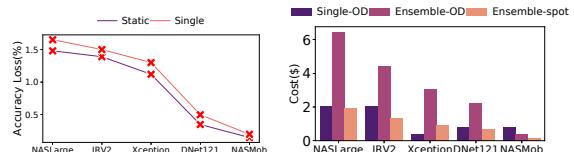
2.3.1 Ensembling Compared to Single Models

To analyze the accuracy offered by ensemble models, we conduct an experiment using 10000 images from ImageNet [29] test dataset, on a C5.xlarge [8] instances in AWS EC2 [5]. For a given baseline model, we combine all models whose latency is lower than that of the baseline, and call it full-ensemble. We perform ensembling on the predictions using a simple majority voting policy. The latency numbers for the baseline models and the corresponding ensemble models along with the size of the ensemble are shown in Table 3. In majority voting, every model votes for a prediction for each input, and the final output prediction is the one that receives more than half of the votes. Figure 3a, shows the accuracy comparison of the baseline (single) and static ensemble (explained in Section 3) compared to the full-ensemble. It is evident that full-ensemble can achieve up to 1.65% better accuracy than single models.

Besides accuracy again, ensembling can also achieve lower latency. The latency of the ensemble is calculated as the time between start and end of the longest running model. As shown in Table 3, in the case of NASLarge, the ensemble latency is 2× lower (151ms) than the baseline latency (311ms). Even a 10ms reduction in latency is of significant importance to the providers [35]. We observe a similar trend of higher ensemble accuracy for other four baseline models with a latency reduction of up to 1.3×. Thus, depending on the model subset used in the ensemble, it achieves better accuracy than the baseline at lower latencies. Note that in our example model-set, the benefits of ensembling will diminish for lower



(a) Accuracy loss compared to full-ensemble.



(b) Cost of full-ensembling hosted on OD and Spot instances.

Figure 3: Cost and accuracy of ensembling vs single models.

accuracies (< 75%) because single models can reach those accuracies. Hence, based on the user constraints, *Cocktail* chooses between ensemble and single models.

2.3.2 Ensembling Overhead

While ensembling can boost accuracy with low latency, their distinctive resource hungry nature drastically increases the deployment costs when compared to single models. This is because more VMs or containers have to be procured to match the resource demands. However, note that the “Packing factor” (P_f) for each model also impacts the deployment costs. P_f in this context is defined as the number of inferences that can be executed concurrently in a single instance without violating the inference latency (on average). Table 1 provides the P_f for 11 different models when executed on a C5.xlarge instance. There is a linear relationship between P_f and the instance size. It can be seen that smaller models (MNet, NASMob) can be packed 2-5× more when compared to larger models (IRV2, NASLarge). Thus, the ensembles with models of higher P_f have significantly lower cost.

The benefits of P_f is contingent upon the models chosen by the model selection policy. Existing ensemble model selection policies used in systems like Clipper use all off-the-shelf models and assign weights to them to calculate accuracy. However, they do not right-size the model selection to include models which primarily contribute to the majority voting. We compare the cost of hosting ensembles using both spot (ensemble-spot) and OD (ensemble-OD) instances with the single models hosted on OD (single-OD) instances. Ensemble-spot is explained further in the next section. We run the experiment over a period of 1 hour for 10 requests/second. The cost is calculated as the cost per hour of EC2 c5.xlarge instance use, billed by AWS [5]. We ensure all instances are fully utilized by packing multiple requests in accordance to the P_f . As shown in Figure 3b, Ensemble-OD is always expensive than single-OD for all the models. Therefore, it is important to ensemble an “optimal” number of less compute intensive models to reduce the cost.

3 Prelude to Cocktail

To specifically address the cost of hosting an ensembling-based model-serving framework in public clouds without sacrificing the accuracy, this section introduces an overview of the two primary design choices employed in *Cocktail*.

How to reduce resource footprint? The first step towards making model ensembling cost effective is to minimize the

number of models by pruning the ensemble, which reduces the overall resource footprint. In order to estimate the right number of models to participate in a given ensemble, we conduct an experiment where we chose top $\frac{N}{2}$ accurate models (static) from the full-ensemble of size N . From Figure 3a, it can be seen that the static policy has an accuracy loss of up to 1.45% when compared to full-ensemble, but is still better than single models. This implies that the models other than top $\frac{N}{2}$ yields a significant 1.45% accuracy improvement in the full-ensemble but they cannot be statically determined.

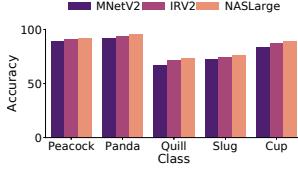


Figure 4: Class-wise Accuracy.

Therefore, a full-ensemble model participation is not required for all the inputs because, every model is individually suited to classify certain classes of images when compared to other classes. Figure 4 shows the class-wise accuracy for three models on 5 distinct classes. It can be seen that for simpler classes like Slug, MNetV2 can achieve similar accuracy as the bigger models, while for difficult classes, like Cup and Quill, it experiences up to 3% loss in accuracy. Since the model participation for ensembling can vary based on the class of input images being classified, there is a scope to develop a dynamic model selection policy that can leverage this class-wise variability to intelligently determine the number of models required for a given input.

Key Takeaway: Full ensemble model-selection is an overkill, while static-ensemble leads to accuracy loss. This calls for a dynamic model selection policy which can accurately determine the number of models required, contingent upon the accuracy and scalability of the model selection policy.

How to save cost? Although dynamic model selection policies can significantly reduce the resource footprint as shown in Figure 3b, the cost is still 20-30% higher when compared to a single model inference. Most cloud providers offer transient VMs such as Amazon Spot instances [69], Google Preemptible VMs [9], and Azure Low-priority VMs [7], that can reduce cloud computing costs by as much as $10\times$ [3]. In *Cocktail*, we leverage these transient VMs such as spot instances to drastically reduce the cost of deploying ensembling model framework. As an example, we host full-ensembling on AWS spot instances. Figure 3b shows that ensemble-spot can reduce the cost by up to $3.3\times$ when compared to ensemble-OD. For certain baselines like IRV2, ensemble-spot is also $1.5\times$ cheaper than single-OD. However, the crucial downside of using transient VMs is that they can be unilaterally preempted by the cloud provider at any given point due to reasons like increase in bid-price or provider-induced random interruptions. As we will discuss further, *Cocktail* is resilient to instance failures owing to the fault-tolerance of ensembling by computing multiple inferences for a single request.

Key takeaway: The cost-effectiveness of transient instances, is naturally suitable for hosting ensemble models.

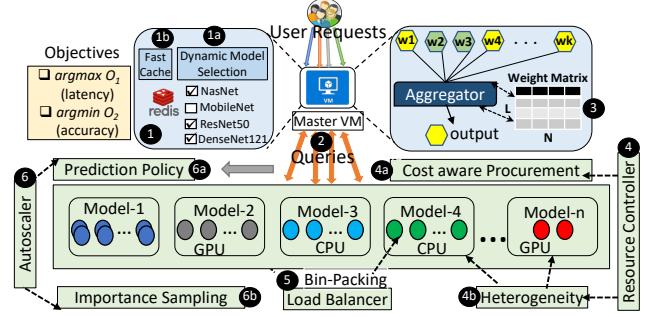


Figure 5: High-level overview of *Cocktail* design.

4 Overall Design of Cocktail

Motivated by our observations, we design a novel model-serving framework, *Cocktail*, that can deliver high-accuracy and low-latency predictions at reduced cost. Figure 5 depicts the high-level design of *Cocktail*. Users submit requests to a master VM, which runs a model selection algorithm, 1a to decide the models to participate in the ensemble. The participating models are made available in a model cache 1b for faster access and avoid re-computation for requests having similar constraints. Then, individual queries are dispatched to instances pools 2 dedicated for each model. The results from the workers are *ensembled* using an weighted majority voting aggregator 3 to agree upon a correct prediction. To efficiently address the resource management and scalability challenges, *Cocktail* applies multiple strategies.

First, it maintains dedicated instance pools to serve individual models which simplifies the management and load balancing overheads for every model. Next, the resource controller 4 handles instance procurement, by exploiting both CPU and GPU instances 4a in a cost-aware 4b fashion, while the load balancer 5 ensures all procured instances are bin-packed by assigning queries to appropriate instances. We also design an autoscaler 6, which utilizes a prediction policy 6a to forecast the request load and scale instances for every model pool, thereby minimizing over-provisioning of resources. The autoscaler further employs an importance sampling 6b algorithm to estimate the importance of each model pool by calculating percentage of request served by it in a given time interval. The key components of the design are explained in detail below.

4.1 Dynamic Model Selection Policy

We use a window-based dynamic model selection policy using two objective functions as described below.

Objective functions: In order to reduce cost and latency while maximizing the accuracy, we define a latency-accuracy metric (μ_{AL}) and cost metric (μ_c):

$$\mu_{AL} = \frac{Acc_{target}}{Lat_{target}} \quad \mu_c = k \times \sum_{m=1}^N \frac{inst_cost}{P_{f_m}}$$

where N is the number of models used to ensemble and $inst_cost$ is the VM cost. Each model m has a packing factor

P_{fm} and k is a constant which depends on the VM size in terms of vCPUs (xlarge, 2xlarge, etc). Our first objective function (O_1) is to maximize μ_{AL} such that target accuracy (Acc_{target}) is reached within the target latency (Lat_{target}).

$$\max \mu_{AL} : \left\{ \begin{array}{l} Acc_{target} \geq Acc_{target} \pm Acc_{margin} \\ Lat_{target} \leq Lat_{target} \pm Lat_{margin} \end{array} \right.$$

To solve O_1 , we determine an initial model list by choosing the individual models satisfying Lat_{target} and then create a probabilistic ensemble that satisfies the Acc_{target} . *Cocktail* takes the accuracy of each model as a probability of correctness and then iteratively constructs a model list, where the joint probability of them performing the classification is within the accuracy target. We tolerate a 0.2% (Acc_{margin}) and 5ms (Lat_{margin}) variance in Acc_{target} and Lat_{target} , respectively. Next, we solve for the second objective function (O_2) by minimizing μ_C , while maintaining the target accuracy.

$$\min \mu_C : \left\{ \begin{array}{l} Acc_{target} \geq Acc_{target} \pm Acc_{margin} \end{array} \right.$$

(O_2) is solved by resizing the model list of size N and further through intelligence resource procurement (described in section 4.2), and thus maximizing P_f and minimizing k simultaneously. For N models, where each model has a minimum accuracy ‘ a ’, we model the ensemble as a coin-toss problem, where N biased coins (with probability of head being a) are tossed together, and we need to find the probability of majority of them being heads. For this, we need at least $\lfloor \frac{N}{2} \rfloor + 1$ models to give the same results. The probability of correct prediction is given by

$$\sum_{i=\lfloor \frac{N}{2} \rfloor + 1}^N \binom{N}{i} a^i (1-a)^{(N-i)}$$

Model Selection Algorithm: To minimize μ_C , we design a policy to downscale the number of models, if more than $N/2+1$ models vote for the same classification result. Algorithm 1 describes the overall design of the model selection policy ①. For every monitoring interval, we keep track of the accuracy obtained from predicting all input images within the interval. If the accuracy of the interval reaches the threshold accuracy (target + error_margin), we scale down the number of available models in the ensemble. For consecutive sampling intervals, we calculate the Mode (most frequently occurring) of the majority vote received for every input. If the Mode is greater than needed votes $\lfloor N/2 \rfloor + 1$ we prune the models to $\lfloor N/2 \rfloor + 1$. While down-scaling, we drop the models with the least prediction accuracy in that interval. If there is a tie, we drop the model with least packing factor (P_f). It can so happen that dropping models can lead to drop in accuracy for certain intervals, because the class of images being predicted are different. In such cases, we up-size the models (one at a time) by adding most accurate model from the remaining unused models.

Algorithm 1 Model Selection and Weighted Majority Voting

```

1: procedure FULL_ENSEMBLE(MODELLIST, SLO)
2:   for model ∈ ModelList do
3:     if model.latency ≤ SLO.latency then
4:       Model.add(model)
5:     end if
6:   end for (O1)
7: end procedure
8: procedure DYNAMIC_MODEL_SCALING(Models)
9:   if curr_accuracy ≥ accuracy_threshold then
10:    if max_vote >  $\frac{N}{2} + 1$  then (O2)
11:      to_be_dropped ← max_vote -  $\frac{N}{2} + 1$ 
12:      Models.drop(to_be_dropped)
13:    end if
14:    else
15:      addModel ← find_models(remaining_models)
16:      Models.append(addModel)
17:    end if
18: end procedure
19: procedure WEIGHTED_VOTING(Models)
20:   for model in ∀Models do
21:     class ← model.predicted_class
22:     weighted_vote[class] += weights[model.class]
23:   end for
24:   Pclass ← max(weighted_vote, key = class)
25:   return Pclass
26: end procedure

```

4.1.1 Class-based Weighted Majority Voting

The model selection policy described above ensures that we only use the necessary models in the majority voting. In order to increase the accuracy of majority voting, we design a weighted majority voting policy ③. The weight matrix is designed by considering the accuracy of each model for each class, giving us a weight matrix of $L \times N$ dimension, where L is the number of unique labels and N is the number of models used in the ensemble. The majority vote is calculated as a sum of model-weights for each unique class in the individual prediction of the ensemble. For instance, if there are 3 unique classes predicted by all the ensemble models, we sum the weights for all models of the same class. The class with the maximum weight (P_{class}) is the output of the majority vote. Hence, classes that did not get the highest votes can still be the final output if the models associated with that class has a higher weight, than the combined weights of highest voted class. Unlike commonly used voting policies which assign weights based on overall correct predictions, our policy incorporates class-wise information to the weights, thus making it more adaptable to different images classes.

In order to determine the weight of every class, we use a per-class dictionary that keeps track of the correct predictions of every model per class. We populate the dictionary at runtime to avoid any inherent bias that could result from varying images over time. Similarly, our model selection policy is also changed at runtime based on correct predictions seen during every interval. An important concern in majority voting is tie-breaking. Ties occur when two sets of equal number of models predict a different result. The effectiveness

Algorithm 2 Predictive Weighted Instance Auto Scaling

```

1: procedure WEIGHTED_AUTOSCALING(Stages)
2:   Predicted_load  $\leftarrow$  DeepARN_Predict(load)
3:   for every Interval do
4:     for model in  $\forall$ Models do
5:       model_weight  $\leftarrow$  get_popularity(model)
6:       Weight.append(model_weight)
7:     end for
8:   end for
9:   if Predicted_load  $\geq$  Current_load then
10:    for model in  $\forall$ Models do
11:       $I_n \leftarrow (Predicted\_load - Current\_load) \times model\_weight$ 
12:      launch_workers(est_VMs)
13:      model.workers.append(est_VMs)
14:    end for
15:   end if
16: end procedure

```

of weighted voting in breaking ties is discussed in Section 6.

4.2 Resource Management

Besides model selection, it is crucial to design an optimized resource provisioning and management scheme to host the models cost-effectively. We explain in detail the resource procurement and autoscaling policy employed in *Cocktail*.

4.2.1 Resource Controller

Resource controller determines the cost-effective combination of instances to be procured. We explain the details below.

Resource Types: We use both CPU and GPU instances ④a depending on the request arrival load. GPU instances are cost-effective when packed with a large batch of requests for execution. Hence, inspired from prior work [27, 86], we design an adaptive packing policy such that it takes into account the number of requests to schedule at time T and P_f for every instance. The requests are sent to GPU instances only if the load matches the P_f of the instance.

Cost-aware Procurement: The cost of executing in a fully packed instance determines how expensive is each instance. Prior to scaling-up instances, we need to estimate the cost ④b of running them along with existing instances. At any given time T , based on the predicted load (L_p) and running instances R_N , we use a cost-aware greedy policy to determine the number of additional instances required to serve as $A_n = L_p - C_r$, where $C_r = \sum_{i=1}^N P_{f_i}$, is the request load which can be handled with R_N . To procure A_n instances, we greedily calculate the least cost instance as $\min_{i \in instances} Cost_i \times A_n / P_{f_i}$. Depending on the cost-effectiveness ratio of A_n / P_{f_i} , GPUs will be preferred over CPU instances.

Load Balancer: Apart from procuring instances, it is quintessential to design a load balancing and bin-packing ⑤ strategy to fully utilize all the provisioned instances. We maintain a request queue at every model pool. In order to increase the utilization of all instances in a pool at any given time, the load balancer submits every request from the queue to the least remaining free slots (viz. instance packing factor P_f). This is similar to an online bin-packing algorithm. We use an idle-timeout limit for 10 minutes to recycle unused

instances from every model pool. Hence, greedily assigning requests enables early scale down of lightly loaded instances.

4.2.2 Autoscaler

Along with resource procurement, we need to autoscale instances to satisfy the incoming query load. Though reactive policies (used in Clipper and InFaas) can be employed which take into account metrics like CPU utilization [83], these policies are slow to react when there is dynamism in request rates. Proactive policies with request prediction are known to have superior performance [86] and can co-exist with reactive policies. In *Cocktail*, we use a load prediction model that can accurately forecast the anticipated load for a given time interval. Using the predicted load ⑥a, *Cocktail* spawns additional instances, if necessary, for every instance pool. In addition, we sample SLO violations for every 10s interval and reactively spawn additional instances to every pool based on aggregate resource utilization of all instances. This captures SLO violations due to mis-predictions.

Prediction Policy: To effectively capture the different load arrival patterns,

Model	RMSE
MWA	77.5
EWMA	88.25
Linear R.	87.5
Logistic R.	78.34
Simple FF.	45.45
LSTM	28.56
DeepArEst	26.67

Table 4: Prediction models.

we design a DeepAR-estimator (DeepARest) based prediction model. We zeroed in on the choice of using DeepARest by conducting (Table 4) an in-depth comparison of the accuracy loss when compared with other state-of-the-art traditional and ML-based prediction models used in prior works [47, 86]. As shown in Algorithm 2, for every model under a periodic scheduling interval of 1 minute (T_s), we use the *Predicted_load* (L_p) at time $T + T_p$ and compare it with the *current_load* to determine the number of instances (I_n). T_p is defined as the average launch time for new instances. (T_s) is set to 1 minute as it is the typical instance provisioning time for EC2 VMs. To calculate (L_p), we sample the arrival rate in adjacent windows of size W over the past S seconds. Using the global arrival rate from all windows, the model predicts (L_p) for T_p time units from T . T_p is set to 10 minutes because it is sufficient time to capture the variations in long-term future. All these parameters are tunable based on the system needs.

Importance Sampling: An important concern in autoscaling is that the model selection policy dynamically determines the models in the ensemble for a given request constraints. Autoscaling the instances equally for every model based on predicted load, would inherently lead to over-provisioned instances for under-used models. To address this concern, we design a weighted autoscaling policy which intelligently auto-scales instances for every pool based on the weights. As shown in Algorithm 2, weights are determined by frequency in which a particular model is chosen for requests (*get_popularity*) with respect to other models in the ensemble.

The weights are multiplied with the predicted load to scale instances (*launch_workers*) for every model pool. We name this as an importance sampling 6b technique, because the model pools are scaled proportional to their popularity.

5 Implementation and Evaluation

We implemented a prototype of *Cocktail* and deployed it on AWS EC2 [5] platform. The details of the implementation are described below. *Cocktail* is open-sourced at <https://github.com/jashwantraj92/cocktail>

5.1 Cocktail Prototype Implementation

Cocktail is implemented using 10KLOC of Python. We designed *Cocktail* as a client-server architecture, where one master VM receives all the incoming requests which are sent to individual model worker VMs.

Master-Worker Architecture: The master node handles the major tasks such as (i) concord model selection policy, (ii) request dispatch to workers VMs as asynchronous future tasks using Python `asyncio` library, and (iii) ensembling the prediction from the worker VMs. Also all VM specific metrics such as `current_load`, CPU utilization, etc. reside in the master node. It runs on a C5.16x [8] large instance to handle these large volume of diverse tasks. Each worker VMs runs a client process to serve its corresponding model. The requests are served as independent parallel threads to ensure timely predictions. We use Python Sanic web-server for communication with the master and worker VMs. Each worker VM runs tensorflow-serving [60] to serve the inference requests.

Load Balancer: The master VMs runs a separate thread to monitor the importance sampling of all individual model pools. It keeps track of the number of requests served per model in the past 5 minutes. This information is used for calculating the weights per model for autoscaling decisions. We integrate a `mongodb` [21] database in the master node to maintain all information about procured instances, spot-instance price list, and instance utilization. The load prediction model resides in the master VM which constantly records the arrival rate in adjacent windows. Recall that the details of the prediction were described in Section 4.2.2. The DeepAREst [4] model was trained using Keras [22] and Tensorflow, over 100 epochs with 2 layers, 32 neurons and a batch-size of 1.

Model Cache: We keep track of the model selected for ensembling on a per request constraint basis. The constraints are defined as `<latency, accuracy>` pair. The queries arriving with similar constraints can read the model cache to avoid re-computation for selecting the models. The model cache is implemented as a hash-map using Redis [16] in-memory key-value store for fast access.

Constraint specification: We expose a simple API to developers, where they can specify the type of inference task (e.g., classification) along with the `<latency, accuracy>` constraints. Developers also need to indicate the primary objective between these two constraints. *Cocktail* automatically

Dataset	Application	Classes	Train-set	Test-set
ImageNet [29]	Image	1000	1.2M	50K
CIFAR-100 [50]	Image	100	50K	10K
SST-2 [72]	Text	2	9.6K	1.8K
SemEval [66]	Text	3	50.3K	12.2K

Table 5: Benchmark Applications and datasets.

chooses a set of single or ensemble models required to meet the developer specified constraints.

Discussion: Our accuracy and latency constraints are limited to the measurements from the available pretrained models. Note that changing the models or/and framework would lead to minor deviations. While providing latency and top-1% accuracy of the pretrained models is an offline step in *Cocktail*, we can calculate these values through one-time profiling and use them in the framework. All decisions related to VM autoscaling, bin-packing and load-prediction are reliant on the centralized `mongodb` database, which can become a potential bottleneck in terms of scalability and consistency. This can be mitigated by using fast distributed solutions like Redis [16] and Zookeeper [46]. The DeepAREst model is pre-trained using 60% of the arrival trace. For varying load patterns, the model parameters can be updated by re-training in the background with new arrival rates.

5.2 Evaluation Methodology

We evaluate our prototype implementation on AWS EC2 [8] platforms. Specifically, we use C5.xlarge, 2xlarge, 4xlarge, 8xlarge for CPU instances and p2.xlarge for GPU instances.

Load Generator: We use different traces which are given as input to the load generator. Firstly, we use real-world request arrival traces from Wikipedia [76], which exhibit typical characteristics of ML inference workloads as it has recurring diurnal patterns. The second trace is production twitter [48] trace which is bursty with unexpected load spikes. We use the first 1 hour sample of both the traces and they are scaled to have an average request rate of 50 req/sec.

Workload: As shown in Table 5 we use image-classification and Sentiment Analysis (text) applications with two datasets each for our evaluation. Sentiment analysis outputs the sentiment of a given sentence as positive negative and (or) neutral. We use 9 different prominently used text-classification models from transformers library [81] (details available in appendix) designed using Google BERT [30] architecture trained on SST [72] and SemEval [66] dataset. Each request from the load-generator is modelled after a query with specific `<latency, accuracy>` constraints. The queries consist of images or sentences, which are randomly picked from the test dataset. In our experiments, we use five different types of these constraints.

As an example for the `Imagenet` dataset shown in Figure 6, each constraint is a representative of `<latency, accuracy>` combination offered by single models (shown in Table 1). We use one constraint (blue dots) each from five different regions

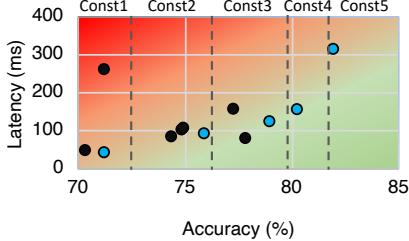


Figure 6: Constraints used in our workloads.

(categorized by dotted lines) picked in the increasing order of accuracy. Each of these picked constraints (named const1 - const5 in the Figure) represents a single baseline model, whose corresponding ensemble size ranges from small (2) to large (10), as shown in Table 3. Note that the latency is the raw model execution latency, and does not include the additional network-transfer overheads incurred. We picked the constraints using a similar procedure by ordering constraints across five different categories for CIFAR-100, SST-2 and SemEval (twitter tweets) datasets. The list of models used for them are given in the Appendix. We model two different workload mixes by using a combination of these five query constraint types. Based on the decreasing order of accuracy, we categorize them into *Strict* and *Relaxed* workloads.

5.2.1 Evaluation Metrics

Most of our evaluations of *Cocktail* for image-classification are performed using the Imagenet dataset. To further demonstrate the sensitivity of *Cocktail* to dataset and applicability to other classification applications, we also evaluate it using CIFAR-100 and Sentiment-Analysis application. We use three important metrics: response latency, cost and accuracy for evaluating and comparing our design to other state-of-the-art systems. The response latency metric includes model inference latency, communication/network latency and synchronization overheads. Queries that do not meet response latency requirements ($>700\text{ms}$) are considered as SLO violations. The cost metric is the billing cost from AWS, and the accuracy metric is measured as the percentage of requests that meet the target accuracy requirements.

We compare these metrics for *Cocktail* against (i) *InFaas* [83], which is our baseline that employs single model selection policy; (ii) *Clipper* [27], which uses static full model selection policy (analogous to AWS AutoGluon); and (iii) *Clipper-X* which is an enhancement to *Clipper* with a simple model selection (drop one model at a time) that does not utilize the mode-based policy enforced in *Cocktail*. Both *InFaas* and *Clipper* share *Cocktail's* implementation setup to ensure a fair comparison with respect to our design and execution environment. For instance, both *Clipper* and *InFaas* employ variants of a reactive autoscaler as described in Section 4.2.2. However, in our setup, both benefit from the distributed autoscaling and prediction policies, thus eliminating variability. Also note that *InFaas* is deployed using OnDemand instances, while both *Clipper* and *Cocktail* use spot instances.

6 Analysis of Results

This section discusses the experimental results of *Cocktail* using the Wiki and Twitter traces. To summarize the overall results, *Cocktail* providing $2\times$ reduction in latency, while meeting the accuracy for up-to 96% of the requests under reduced deployment cost by $1.4\times$, when compared to *InFaas* and *Clipper*.

6.1 Latency, Accuracy and Cost Reduction

Latency Distribution: Figure 7 shows the distribution of total response latency in a standard box-and-whisker plot. The boundaries of the box-plots depict the 1st quartile (25th percentile (PCTL)) and 3rd quartile (75th PCTL), the whiskers plot the minimum and maximum (tail) latency and the middle line inside the box depict the median (50 PCTL). The total response latency includes additional 200-300ms incurred for query serialization and data transfer over network. It can be seen that the maximum latency of *Cocktail* is similar to the 75th PCTL latency of *InFaas*. This is because the single model inference have up to 2x higher latency to achieve higher accuracy. Consequently, this leads to 35% SLO violations for *InFaas* in the case of Strict workload. In contrast, both *Cocktail* and *Clipper* can reach the accuracy at lower latency due to ensembling, thus minimizing SLO violations to 1%.

Also, the tail latency is higher for Twitter trace (Figure 7c, 7d) owing to its bursty nature. Note that the tail latency of *Clipper* is still higher than *Cocktail* because *Clipper* ensembles more models than *Cocktail*, thereby resulting in straggler tasks in the VMs. The difference in latency between *Cocktail* and *InFaas* is lower for Relaxed workload when compared to Strict workload (20% lower in tail). Since the Relaxed workload has much lower accuracy constraints, smaller models are able to singularly achieve the accuracy requirements at lower latency.

Accuracy violations: The accuracy is measured as a moving window average with size 200 for all the requests in the workload.

Both *Clipper* and *Cocktail* can meet the accuracy for 56% of requests, which is 26% and 9% more than *InFaas* and *Clipper* respectively. This is because, intuitively ensembling leads to higher accuracy than single models.

However, *Cocktail* is still 9% better than *Clipper* because the class-based weighted voting, is efficient in breaking ties when compared to weighting averaging used in *Clipper*. Since majority voting can include ties in votes, we analyzed the number of ties, which were correctly predicted for all the queries. *Cocktail* was able to deliver correct predictions for 35% of the tied votes, whereas breaking the ties in *Clipper* led only to 20% correct predictions.

Scheme	Accuracy Met (%)	
	Strict	Relaxed
<i>InFaas</i>	21	71
<i>Clipper</i>	47	89
<i>Cocktail</i>	56	96

Table 6: Requests meeting target accuracy averaged for both Trace.

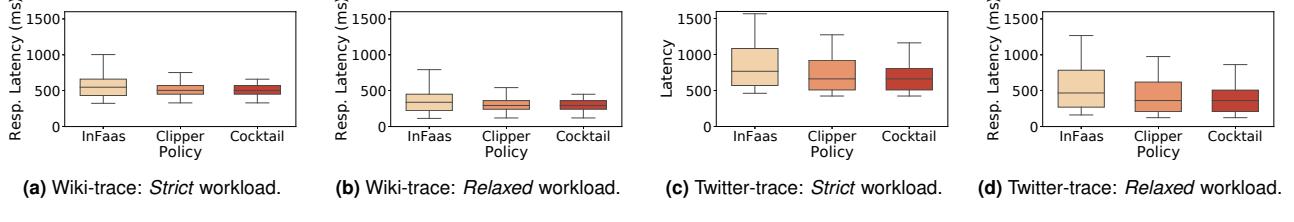


Figure 7: Latency Distribution of *InFaas*, *Clipper* and *Cocktail* for two workload mixes using both Wiki and Twitter traces.

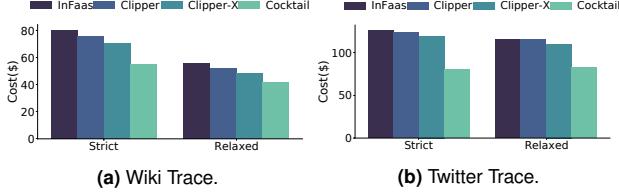


Figure 8: Cost savings of *Cocktail* compared to three schemes.

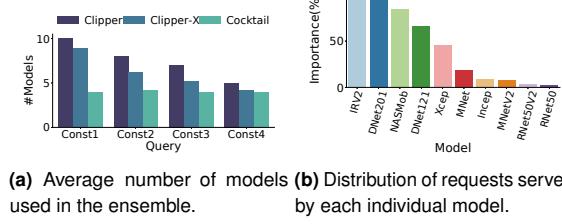


Figure 9: Benefits of dynamic model selection policy.

Note that, changing the target accuracy to tolerate a 0.5% loss, increases the percentage of requests that meet accuracy to 81% for *Cocktail*, when compared to 61% for *InFaas*. The requests meeting accuracy are generally higher for the *Relaxed* workload because the target accuracy is much lower. Overall, *Cocktail* was able to deliver an accuracy of 83% and 79.5% on average for the *Strict* and *Relaxed* workloads, respectively. This translates to 1.5% and 1% better accuracy than *Clipper* and *InFaas*. We do not plot the results for *Clipper-X*, which achieves similar accuracy to *Cocktail*, but uses more models as explained in Section 6.2.1.

Cost Comparison: Figure 8 plots the cost savings of *Cocktail* when compared to *InFaas*, *Clipper* and *Clipper-X* policies. It can be seen that, *Cocktail* is up to $1.45 \times$ more cost effective than *InFaas* for *Strict* workload. In addition, *Cocktail* reduces cost by $1.35 \times$ and $1.27 \times$ compared to *Clipper* and *Clipper-X* policies, owing to its dynamic model selection policy, which minimizes the resource footprint of ensembling. On the other hand, *Clipper* uses all models in ensemble and the *Clipper-X* policy does not right size the models as aggressively as *Clipper*, hence they are more expensive. Note that, all the schemes incur higher cost for twitter trace (Figure 8b) compared to wiki trace (Figure 8a). This is because the twitter workload is bursty, thereby leading to intermittent over-provisioned VMs.

6.2 Key Sources of Improvements

The major improvements in terms of cost, latency, and accu-

racy in *Cocktail* are explained below. For brevity in explanation, the results are averaged across Wiki and Twitter traces for strict workload.

6.2.1 Benefits from dynamic model selection

Figure 9a plots the average number of models used for queries falling under the first four different constraint (const) types. Here, *Cocktail* reduces the number of models by up to 55% for all four query types. This is because our dynamic policy ensures that the number of models are well within $N/2$ most of the time, whereas the *Clipper-X* policy does not aggressively scale down models. *Clipper*, on the other hand, is static and always uses all the models. The percentage of model-reduction is lower for *Const2*, 3 and 4 because, the total models used in the ensemble is less than *Const1* (8, 7 and 6 models, respectively). Still, the savings in terms of cost will be significant because even removing one model from the ensemble amounts to $\sim 20\%$ cost savings in the long run (*Clipper* vs *Clipper-X* ensemble in Figure 8). Thus, the benefits of *Cocktail* are substantial for large ensembles while reducing the number of models for medium-sized ensembles.

Figure 9b shows the breakdown of the percentage of requests (*Const1*) served by the each model. As seen, InceptionResNetV2, Densenet-201, Densenet121, NasnetMobile and Xception are the top-5 most used models in the ensemble. Based on Table 1, if we had statically taken the top $N/2$ most accurate models, NasNetmobile would not have been included in the ensemble. However, based on the input images sent in each query, our model selection policy has been able to identify NasNetMobile to be a significantly contributing model in the ensemble. Further, the other 5 models are used by up to 25% of the images. Not including them in the ensemble would have led to severe loss in accuracy. But, our dynamic policy with the class-based weighted voting, adapts to input images in a given interval by accurately selecting the best performing model for each class. To further demonstrate the effectiveness of our dynamic model selection,

Figure 10b,10c plots the number models in every sampling interval along with cumulative accuracy and window accuracy within each sampling interval for three schemes. We observe that *Cocktail* can effectively scale up and scale down the models while maintaining the cumulative accuracy well within the threshold. More than 50% of the time the number of models are maintained between 4 to 5, because the dynamic policy is quick in detecting accuracy failures and recovers immediately

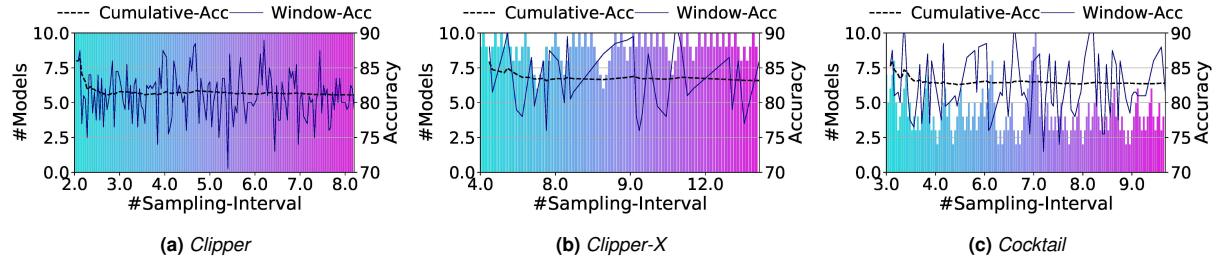


Figure 10: Figures (a), (b) and (c) shows the number of models used in ensemble with corresponding cumulative accuracy and window accuracy over a 1 hour period for requests under *Const1*. Figure (d) shows the effects of distributed autoscaling with importance sampling.

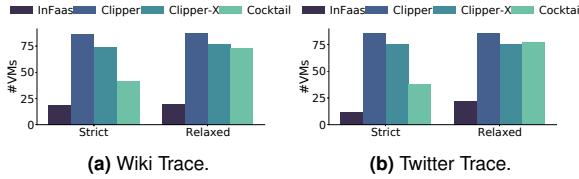


Figure 11: Number of VMs spawned for all four schemes.

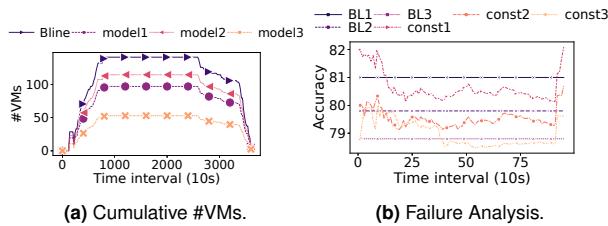


Figure 12: Sensitivity analysis of VMs.

by scaling up models. However, *Clipper-X* does not scale down models as frequently as *Cocktail*, while ensuring similar accuracy. *Clipper* is less accurate than *Cocktail* and further it uses all 10 models throughout.

6.2.2 Benefits from Autoscaling

Figure 11 plots the reduction in the number of VMs used by all four schemes. It can be seen that both *Cocktail* and *Clipper-X* spawn 49% and 20% fewer VMs than *Clipper* for workload-1 on Twitter trace. *Cocktail* spawns 29% lesser VMs on top of *Clipper-X*, because it is not aggressive enough like *Cocktail* to downscale more models at every interval. It is to be noted that the savings are lower for *Relaxed* workload because, the number of models in the ensemble are inherently low, thus leading to reduced benefits from scaling down the models. Intuitively, *InFaas* has the least number of VMs spawned because it does not ensemble models. *Cocktail* spawns upto 50% more VMs than *InFaas*, but in turns reduces accuracy loss by up to 96%.

To further capture the benefits of the weighted autoscaling policy, Figure 12a plots the number of VMs spawned over time for the top-3 most used models in the ensemble for *Const1*. The Bline denotes number of VMs that would be spawned without applying the weights. Not adopting an importance sampling based weighted policy would result in equivalent number of VMs as the Bline for all models. However, since *Cocktail* exploits importance sampling by keeping track of the frequency in which models are selected, the num-

ber of VMs spawned for model1, model2 and model-3 is upto 3× times lesser than uniform scaling. Figure 9b shows the most used models in decreasing order of importance. The autoscaling policy effectively utilizes this importance factor in regular intervals of 5 minutes. Despite using multiple models for a single inference, importance sampling combined with aggressive model pruning, greatly reduces the resource footprint which directly translates to the cost savings in *Cocktail*.

6.2.3 Benefits of Transient VMs

The cost-reductions in *Cocktail* are akin to cost-savings of transient VMs compared to On-Demand (OD) VMs. We profile the spot price of 4 types of C5 EC2 VMs over a 2-week period in August 2020. It was seen that, the spot instance prices have predictable fluctuations. When compared to the OD price, they were up to 70% cheaper. This price gap is capitalized in *Cocktail* to reduce the cost of instances consumed by ensembling. Note that, we set the bidding price conservatively to 40% of OD. Although, *Cocktail* spawns about 50% more VMs than *InFaas*, the high P_f of small models and spot-instance price reductions combined with autoscaling policies lead to the overall 30-40% cost savings.

6.3 Sensitivity Analysis

In this section, we analyze the sensitivity of *Cocktail* with respect to various design choices which include (i) sampling interval of the accuracy measurements, (ii) spot-instance failure rate and (iii) type of datasets and applications.

6.3.1 Sampling Interval

To study the sensitivity with respect to the sampling interval for measure accuracy loss/gain, we use four different intervals of 10s, 30s, 60s and 120s. Figure 13 plots the average number of models (bar- left y-axis) and cumulative accuracy (line-right y-axis) for the different sampling intervals for queries with three different constraints. It can be seen that the 30s interval strikes the right balance with less than 0.2% loss in accuracy and has average number models much lesser than other intervals. This is because, increasing the interval leads to lower number of scale down operations, thus resulting in a bigger ensemble. As a result, the 120s interval has the highest number of models.

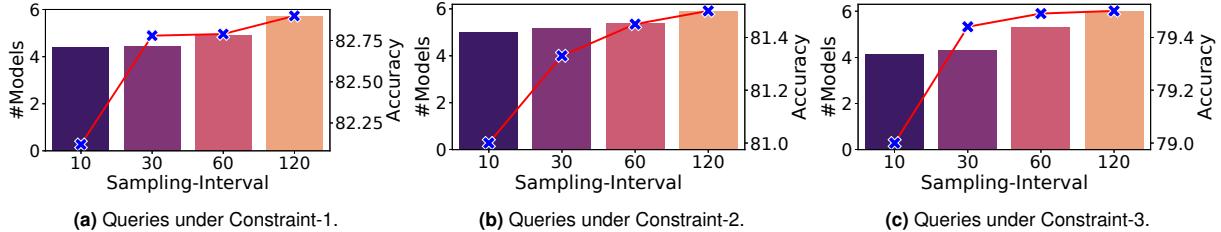


Figure 13: Sensitivity analysis of model selection with respect to sampling interval. The average number of models is in primary axis and cumulative accuracy in secondary axis.

6.3.2 Cocktail Failure Resilience

We use spot instances to host models in *Cocktail*. As previously discussed in Section 3, spot instances interruptions can lead to intermittent loss in accuracy as certain models will be unavailable in the ensemble. However for large ensembles (5 models are more), the intermittent accuracy loss is very low. Figure 12b plots the failure analysis results for top three constraints by comparing the ensemble accuracy to the target accuracy. The desired accuracy for all three constraints are plotted as BL1, BL2 and BL3. We induce failures in the instances using *chaosmonkey* [19] tool with a 20% failure probability. It can be seen that queries in all three constraints suffer an intermittent loss in accuracy of 0.6% between the time period 240s and 800s. Beyond 800s, they quickly recover back to the required accuracy because additional instances are spawned in place of failed instances. However, in the case of *InFaaS*, this would lead to 1% failed requests due to requests being dropped from the failed instances.

An alternate solution would be to restart the queries in running instances but that leads to increased latencies for the 1% requests. In contrast, *Cocktail* incurs a modest accuracy loss of well within 0.6% and quickly adapts to reach the target accuracy. Thus, *Cocktail* is inherently fault-tolerant owing to the parallel nature in computing multiple inferences for a single request. We observe similar accuracy loss or lower for different probability failures of 5%, 10% and 25%, respectively (results/charts omitted in the interest of space).

Discussion: For applications that are latency tolerant, we can potentially redirect requests from failed instances to existing instances, which would lead to increased tail latency. The results we have shown are only for latency intolerant applications. Note that, the ensembles used in our experiments are at-least 4 models or more. For smaller ensembles, instance failures might lead to higher accuracy loss, but in our experiments, single models typically satisfy their constraints.

6.3.3 Sensitivity to Constraints

Figure 14 plots the sensitivity of model selection policy under a wide-range of latency and accuracy constraints. In Figure 14a, we vary the latency under six different constant accuracy categories. It can be seen that for fixed accuracy of 72%, 78% and 80%, the average number of models increase with increase in latency, but drops to 1 for the highest latency. Intuitively, single large models with higher latency can satisfy

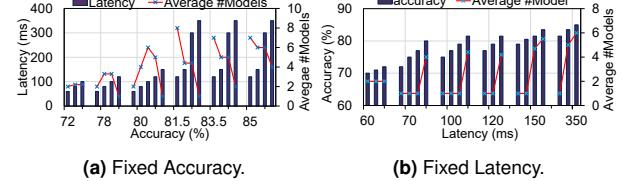


Figure 14: Sensitivity Constraints under fixed latency and accuracy. Bar graphs (latency) plotted using primary y-axis and line graph (#models) plotted using secondary y-axis.

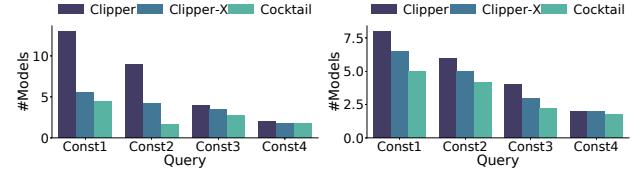


Figure 15: Average number of models used in the ensemble.

the accuracy, while short latency models need to be assembled to reach the same accuracy. For accuracy greater than 80%, the ensemble size drops with higher latencies. This is because the models which offer higher accuracy are typically dense and hence, smaller ensembles are sufficient. In Figure 14b, we vary the accuracy under six different constant latency categories. It can be seen that for higher accuracies, *Cocktail* tries to ensemble more models to reach the accuracy, while for lower accuracy it resorts to using single models.

6.3.4 Sensitivity to Dataset

To demonstrate the applicability of *Cocktail* to multiple datasets, we conducted similar experiments as elucidated in Section 5.2.1 using the CIFAR-100 dataset [50]. It comprises of 100 distinct image classes and we trained 11 different models including the nine that are common from Table 1. Figure 15a plots the average number of models used by the three policies for the top four constraints. It can be seen that *Cocktail* shows similar reduction (as Imagenet) while using only 4.4 models on average. As expected, *Clipper* and *Clipper-X* use more models than *Cocktail* (11 and 5.4, respectively) due to non-aggressive scaling down of the models used.

Figure 16a plots the latency reduction and accuracy boost when compared to *InFaaS* (baseline). While able to reduce 60% of the models used in the ensemble, *Cocktail* also reduces latency by up to 50% and boosts accuracy by up to 1.2%. *Cocktail* was also able to deliver modest accuracy gain

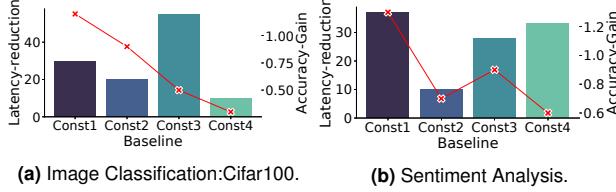


Figure 16: Latency reduction (%) plotted as bar graph(primary y-axis) and accuracy gains (%) plotted as line graph (secondary y-axis) over InFaaS.

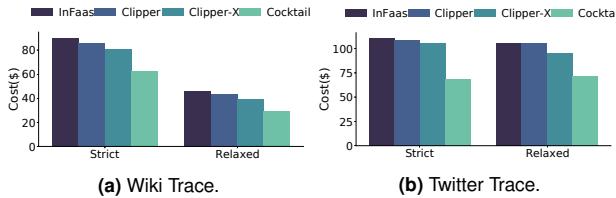


Figure 17: Cost savings of Cocktail for Sentiment Analysis.

of 0.5% than *Clipper* (not plotted). The accuracy gain seen in CIFAR-100 is lesser than ImageNet dataset because the class-based weighted voting works effectively when handling large number of classes (100 in CIFAR vs 1000 in ImageNet). Nevertheless, *Cocktail* is able to deliver the accuracy at 2x lower latency than *InFaaS* and 1.35x lower cost than *Clipper*.

6.4 General Applicability of Cocktail

To demonstrate the general applicability of *Cocktail* to other classification tasks, we evaluated *Cocktail* using a Sentiment Analysis application for two datasets. The results reported are averaged across both the datasets. Figure 15b plots the average number of models used by the three policies for the top four constraints. As shown for *Const1*, *Cocktail* shows similar reduction (as image-classification) with only using 4.8 models on average, which is 40% and 26% lower than *Clipper* and *Clipper-X*, respectively. *Cocktail* is also able to reduce the number of models by 30% and 50% for medium ensembles (*Const2* & *Const3*) as well.

Figure 16b plots the latency reduction and accuracy gain, compared to *InFaaS* (baseline). While being able to reduce 50% of the models used in the ensemble, *Cocktail* also reduces latency by up to 50% and improves accuracy by up to 1.3%. Both *Cocktail* and *Clipper* deliver the same overall accuracy (96%, 94.5%, 93.5%, and 92%). Since sentiment analysis only has 2-3 classes, there are no additional accuracy gains by using the class-based weighted voting. However, the model selection policy effectively switches between different models based on the structure of input text (equivalent to classes in images). For instance, complex sentences are more accurately classified by denser models compared to smaller. Despite the lower accuracy gains, *Cocktail* is able to reduce the cost (Figure 17) of model-serving by 1.45x and 1.37x for Wiki trace compared to *InFaaS* and *Clipper*, respectively.

7 Concluding Remarks

There is an imminent need to develop model serving systems that can deliver highly accurate, low latency predictions at reduced cost. In this paper, we propose and evaluate *Cocktail*, a cost-effective model serving system that exploits ensembling techniques to meet high accuracy under low latency goals. In *Cocktail*, we adopt a three-fold approach to reduce the resource footprint of model ensembling. More specifically, we (i) develop a novel dynamic model selection, (ii) design a prudent resource management scheme that utilizes weighted autoscaling for efficient resource allocation, and (iii) leverage transient VM instances to reduce the deployment costs. Our results from extensive evaluations using both CPU and GPU instances on AWS EC2 cloud platform demonstrate that *Cocktail* can reduce deployment cost by 1.4x, while reducing latency by 2x and satisfying accuracy for 96% of requests, compared to the state-of-the-art model-serving systems.

Acknowledgments

We are indebted to our shepherd Manya Ghobadi, the anonymous reviewers and Anup Sarma for their insightful comments to improve the clarity of the presentation. Special mention to Nachiappan Chidambaram N. for his intellectual contributions. This research was partially supported by NSF grants #1931531, #1955815, #1763681, #1908793, #1526750, #2116962, #2122155, #2028929 ,and we thank NSF Chameleon Cloud project CH-819640 for their generous compute grant. All product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

- [1] Martín Abadi. Tensorflow: learning functions at scale. In *AcM Sigplan Notices*. ACM, 2016.
- [2] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. Laser: A scalable response prediction platform for online advertising. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 173–182, 2014.
- [3] Ahmed Ali-Eldin, Jonathan Westin, Bin Wang, Prateek Sharma, and Prashant Shenoy. Spotweb: Running latency-sensitive distributed web services on transient cloud servers. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 1–12, 2019.
- [4] Amazon. Deepar estimator. <https://docs.aws.amazon.com/sagemaker/latest/dg/deepar.html>, February 2020.
- [5] Amazon. EC2 pricing. <https://aws.amazon.com/ec2/pricing/>.
- [6] Amazon. Sagemaker. <https://aws.amazon.com/sagemaker/>, February 2018.
- [7] Amazon. Azure Low priority batch VMs., February 2018. <https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms> .
- [8] Amazon. EC2 C5 Instances., February 2018. <https://aws.amazon.com/ec2/instance-types/c5/> .
- [9] Amazon. Google Preemptible VMs., February 2018. <https://cloud.google.com/preemptible-vms> .
- [10] Azure. Machine Learning as a Service., February 2018. <https://azure.microsoft.com/en-us/pricing/details/machine-learning-service/> .
- [11] Azure. Ensembling in Azure ML Studio., February 2020. <https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/multiclass-decision-forest> .

- [12] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA, 2019. Association for Computing Machinery.
- [13] Marian Stewart Bartlett, Gwen Littlewort, Mark Frank, Claudia Lainscsek, Ian Fasel, and Javier Movellan. Recognizing facial expression: machine learning and application to spontaneous behavior. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 568–573. IEEE, 2005.
- [14] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine learning*, 36(1-2):105–139, 1999.
- [15] William H Beluch, Tim Genewein, Andreas Nürnberger, and Jan M Köhler. The power of ensembles for active learning in image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9368–9377, 2018.
- [16] Josiah L Carlson. *Redis in action*. Manning Publications Co., 2013.
- [17] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. In *Proceedings of the twenty-first international conference on Machine learning*, page 18, 2004.
- [18] Jesús Cerquides and Ramon López De Mántaras. Robust bayesian linear classifier ensembles. In *European Conference on Machine Learning*, pages 72–83. Springer, 2005.
- [19] Michael Alan Chang, Bredan Tschaen, Theophilus Benson, and Laurent Vanbever. Chaos monkey: Increasing sdn reliability through systematic network destruction. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 371–372, 2015.
- [20] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalml: How to use ml prediction apis more accurately and cheaply. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- [21] Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage.* "O'Reilly Media, Inc.", 2013.
- [22] Francois Fleuret. *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. MITP-Verlags GmbH & Co. KG, 2018.
- [23] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *SoCC*, 2018.
- [24] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.
- [25] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, Zhao Zhang, Michael J Franklin, Ali Ghodsi, and Michael I Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809*, 2014.
- [26] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. Inferline: ML inference pipeline composition framework. *CoRR*, abs/1812.01776, 2018.
- [27] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.
- [28] Deepstudio. Deep Learning Dtudio, February 2020. <https://docs.deepcognition.ai/>.
- [29] J. Deng, W. Dong, R. Socher, L. Li, and and. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, June 2009.
- [30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.
- [31] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data, 2020.
- [32] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Dixin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
- [33] Mikel Galar, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera. A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(4):463–484, 2012.
- [34] Arpan Gujarati, Sameh Elmikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *USENIX Middleware Conference*, 2017.
- [35] Arpan Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Banff, Alberta, November 2020. USENIX Association.
- [36] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C.Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. Fifer: Tackling Resource Underutilization in the Serverless Era. In *USENIX Middleware Conference*, 2020.
- [37] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesisidis, and Chita Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *IEEE CLOUD*, 2019.
- [38] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita R. Das. Towards designing a self-managed machine learning inference serving system inpublic cloud, 2020.
- [39] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G. Wei, H. S. Lee, D. Brooks, and C. Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995, 2020.
- [40] Rui Han, Moustafa M. Ghanem, Li Guo, Yike Guo, and Michelle Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Gener. Comput. Syst.*, 32(C):82–98, March 2014.
- [41] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons. Tributary: spot-dancing for elastic services with latency SLOs. In *ATC*, 2018.
- [42] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. Proteus: Agile ML Elasticity Through Tiered Reliability in Dynamic Resource Markets. In *Eurosys*, 2017.
- [43] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *ASPLOS*, 2015.
- [44] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, Feb 2018.
- [45] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1314–1324, 2019.
- [46] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, 2010.
- [47] Minoru Kawashima, Charles E Dorgan, and John W Mitchell. Hourly thermal load prediction for the next 24 hours by arima, ewma, lr and

- an artificial neural network. Technical report, American Society of Heating, Refrigerating and Air-Conditioning Engineers ..., 1995.
- [48] Abeer Abdel Khaleq and Ilkyeon Ra. Cloud-based disaster management as a service: A microservice approach for hurricane twitter data analysis. In *GHTC*, 2018.
- [49] J Zico Kolter and Marcus A Maloof. Dynamic weighted majority: An ensemble method for drifting concepts. *Journal of Machine Learning Research*, 8(Dec):2755–2790, 2007.
- [50] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-100 (canadian institute for advanced research), 2010. <http://www.cs.toronto.edu/~kriz/cifar.html>.
- [51] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.
- [52] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, Carlsbad, CA, October 2018. USENIX Association.
- [53] Romain Lerallut, Diane Gasselin, and Nicolas Le Roux. Large-scale real-time product recommendation at criteo. In *Proceedings of the 9th ACM Conference on Recommender Systems*, pages 232–232, 2015.
- [54] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.
- [55] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.
- [56] Zhenyu Lu, Xindong Wu, Xingquan Zhu, and Josh Bongard. Ensemble pruning via individual contribution ordering. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’10, page 871–880, New York, NY, USA, 2010. Association for Computing Machinery.
- [57] Cyan Subhra Mishra, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. Origin: Enabling on-device intelligence for human activity recognition using energy harvesting wireless sensor networks. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1414–1419, 2021.
- [58] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. Alembic: Automated model inference for stateful network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 699–718, Boston, MA, February 2019. USENIX Association.
- [59] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.
- [60] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soye. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.
- [61] Nikunj C Oza. Online bagging and boosting. In *2005 IEEE international conference on systems, man and cybernetics*, volume 3, pages 2340–2345. Ieee, 2005.
- [62] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.
- [63] Heyang Qin, Syed Zawad, Yanqi Zhou, Lei Yang, Dongfang Zhao, and Feng Yan. Swift machine learning model serving scheduling: a region based reinforcement learning approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–23, 2019.
- [64] Xueheng Qiu, Le Zhang, Ye Ren, Ponnuthurai N Suganthan, and Gehan Amarasinghe. Ensemble deep learning for regression and time series forecasting. In *2014 IEEE symposium on computational intelligence in ensemble learning (CIEL)*, pages 1–6. IEEE, 2014.
- [65] Atul Rahman, Jongeun Lee, and Kiyoung Choi. Efficient fpga acceleration of convolutional neural networks using logical-3d compute array. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1393–1398. IEEE, 2016.
- [66] Sara Rosenthal, Noura Farra, and Preslav Nakov. SemEval-2017 task 4: Sentiment analysis in Twitter. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 502–518, Vancouver, Canada, August 2017. Association for Computational Linguistics.
- [67] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.
- [68] Prateek Sharma, David Irwin, and Prashant Shenoy. Portfolio-driven resource management for transient cloud servers. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1), June 2017.
- [69] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.
- [70] Steven A Shaya, Neal Matheson, John Anthony Singarayar, Nikiforos Kollias, and Jeffrey Adam Bloom. Intelligent performance-based product recommendation system, October 5 2010. US Patent 7,809,601.
- [71] Richard Socher, Yoshua Bengio, and Chris Manning. Deep learning for nlp. *Tutorial at Association of Computational Logistics (ACL)*, 2012.
- [72] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.
- [73] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.
- [74] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das. Phoenix: A constraint-aware scheduler for heterogeneous datacenters. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017.
- [75] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das. Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters. In *CLUSTER*, 2019.
- [76] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 2009.
- [77] Alexander Vezhnevets and Vladimir Vezhnevets. Modest adaboost-teaching adaboost to generalize better. In *Graphicon*, pages 987–997, 2005.
- [78] Jasper A Vrugt and Bruce A Robinson. Treatment of uncertainty using ensemble methods: Comparison of sequential data assimilation and bayesian model averaging. *Water Resources Research*, 43(1), 2007.
- [79] Cheng Wang, Bhuvan Urgaonkar, Neda Nasiriani, and George Kesidis. Using burstable instances in the public cloud: Why, when and how? *SIGMETRICS*, June 2017.
- [80] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. Rafiki: machine learning as an analytics service system. *Proceedings of the VLDB Endowment*, 12(2):128–140, 2018.
- [81] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierrick Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface’s transformers: State-of-

- the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.
- [82] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.
- [83] Neeraja J. Yadwadkar, Francisco Romero, Qian Li, and Christos Kozyrakis. A case for managed and model-less inference serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, New York, NY, USA, 2019. Association for Computing Machinery.
- [84] Tien-Ju Yang, Andrew G. Howard, Bo Chen, Xiao Zhang, Alec Go, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. *CoRR*, abs/1804.03230, 2018.
- [85] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pre-training for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.
- [86] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *ATC*, 2019.
- [87] Honglei Zhuang, Chi Wang, and Yifan Wang. Identifying outlier arms in multi-armed bandit. In *Advances in Neural Information Processing Systems*, pages 5204–5213, 2017.
- [88] Sheikh Ziauddin and Matthew N Dailey. Iris recognition performance enhancement using weighted majority voting. In *2008 15th IEEE International Conference on Image Processing*, pages 277–280. IEEE, 2008.

Appendix

A Modeling of Ensembling

While performing an ensemble it is important to be sure that we can reach the desired accuracy by combining more models. In our design, we solve our first objective function (described in Section 4.1) by combining all available models which meet the latency SLO. To be sure that the combination will give us the desired accuracy of the larger model, we try to theoretically analyse the scenario. We formulate the problem conservatively as following.

We perform an inference by ensembling ‘N’ models, and each of these models have accuracy ‘a’. Therefore the probability of any model giving a correct classification is ‘a’. We assume the output to be correct if majority of them, i.e. $\lfloor N/2 \rfloor + 1$ of them give the same result. Then, the final accuracy of this ensemble would be the probability of at least $\lfloor N/2 \rfloor + 1$ of them giving a correct result.

To we model this problem as a coin-toss problem involving N biased coins with having probability of occurrence of head to be a . Relating this to our problem, each coin represents a model, and an occurrence of head represents the model giving the correct classification. Hence, the problem boils down to find the probability of at least $\lfloor N/2 \rfloor + 1$ heads when all N coins are tossed together. This is a standard binomial distribution problem and can be solved by using the following formula:

$$P_{\text{head}} = \sum_{i=\lfloor \frac{N}{2} \rfloor + 1}^N \binom{N}{i} a^i (1-a)^{(N-i)}.$$

To further quantify, let us consider the case where we need to determine if we can reach the accuracy of NasNetLarge (82%) by combining rest of the smaller models which have lesser latency than NasNetLarge. We have 10 (therefore $N = 10$) such models and among them the least accurate model is MobileNetV1 (accuracy 70%, therefore $a = 0.70$). We need to find the probability of at least 6 of them being correct. Using the equation above we find the probability to be

$$P_{\text{head}} = \sum_{i=\lfloor \frac{10}{2} \rfloor + 1=6}^{10} \binom{10}{i} 0.7^i (1-0.7)^{(10-i)} = 0.83$$

This corresponds to an accuracy of 83%, which is greater than our required accuracy of 82%). Given all the other models have higher accuracy, the least accuracy we can expect with such an ensemble is 83%. This analysis forms the base of our ensemble technique, and hence proving the combination of multiple available models can be more accurate than the most accurate individual model.

B Why DeepARest Model?

We quantitatively justify the choice of using DeepARest by conducting a brick-by-brick comparison of the accuracy loss

when compared with other state-of-the-art prediction models used in prior work.

Table 4 shows the root mean squared error (RMSE) incurred by all the models. The ML models used in these experiments are pre-trained with 60% of the Twitter arrival trace. It is evident that the LSTM and DeepAREst have lowest RMSE value. DeepAREst is 10% better than LSTM model. Since the primary contribution in *Cocktail* is to provide high accuracy and low latency predictions at cheaper cost, application developers can adapt the prediction algorithm to their needs or even plug-in their own prediction models.

C System Overheads

We characterize the system-level overheads incurred due to the design choices in *Cocktail*. The *mongodb* database is a centralized server, which resides on the head-node. We measure the overall average latency incurred due to all reads/writes in the database, which is well within 1.5ms. The DeepAREst prediction model which is not in the critical decision-making path runs as a background process incurring 2.2 ms latency on average. The weighted majority voting takes 0.5ms and the model selection policy takes 0.7ms. The time taken to spawn new VM takes about 60s to 100s depending on the size of the VM instance. The time taken to choose models from the model-cache is less than 1ms. The end-to-end response time to send the image to a worker VM and get the prediction back, was dominated by about 300ms (at maximum) of payload transfer time.

D Instance configuration and Pricing

Instance	vCPUs	Memory	Price
C5a.xlarge	4	8 GiB	\$0.154
C5a.2xlarge	8	16 GiB	\$0.308
C5a.4xlarge	16	32 GiB	\$0.616
C5a.8xlarge	32	64 GiB	\$1.232

Table 7: Configuration and Pricing for EC2 C5 instances.

E CIFAR-100 and BERT Models

Table 8 shows the different models available for image prediction, that are pretrained on Keras using CIFAR-100 dataset.

Model	Params (M)	Top-1 Accuracy(%)	Latency (ms)	P_f
Albert-base [51]	11	91.4	55	7
CodeBert [32]	125	89	79	6
DistilBert [67]	66	90.6	92	5
Albert-large	17	92.5	120	4
XLNet [85]	110	94.6	165	3
Bert [30]	110	92	185	3
Roberta [55]	355	94.3	200	2
Albert-xlarge	58	93.8	220	1
Albert-xxlarge	223	95.9	350	1

Table 9: Pretrained models for Sentiment Analysis using BERT.

Similarly Table 9 shows the different models trained for BERT-based sentiment analysis on twitter dataset.

Model	Params (M)	Top1 Accuracy %	Latency (ms)	P_f
SqueezeNet	4,253,864	70.10	43.45	10
MobileNEt V2	4,253,864	68.20	41.5	10
Inception V4	23,851,784	76.74	74	6
Resnet50	95,154,159	79.20	98.22	5
ResNet18	44,964,665	76.26	35	6
DenseNet-201	20,242,984	79.80	152.21	2
DenseNet-121	8,062,504	78.72	102.35	3
Xception	22,910,480	77.80	119.2	4
NasNet	5,326,716	77.90	120	3
InceptionResnetV2	2,510,000	80.30	251.96	1

Table 8: Pretrained models for CIFAR-100 using Imagenet.

F Spot Instance Price Variation

We profile the spot price of 4 types of C5 EC2 VMs over a 2-week period in August 2020. The price variation is shown in Fig 18.

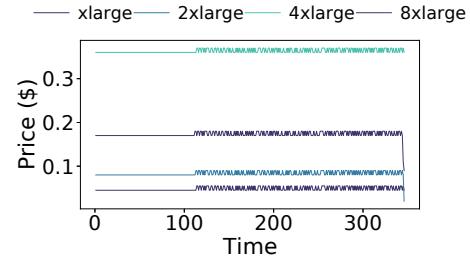


Figure 18: Spot instance price variation (time is in hours).

Data-Parallel Actors: A Programming Model for Scalable Query Serving Systems

Peter Kraft
Stanford University

Fiodar Kazhamiaka
Stanford University

Peter Bailis
Stanford University

Matei Zaharia
Stanford University

Abstract

We present *data-parallel actors (DPA)*, a programming model for building distributed query serving systems. Query serving systems are an important class of applications characterized by low-latency data-parallel queries and frequent bulk data updates; they include data analytics systems like Apache Druid, full-text search engines like ElasticSearch, and time series databases like InfluxDB. They are challenging to build because they run at scale and need complex distributed functionality like data replication, fault tolerance, and update consistency. DPA makes building these systems easier by allowing developers to construct them from purely single-node components while automatically providing these critical properties. In DPA, we view a query serving system as a collection of stateful actors, each encapsulating a partition of data. DPA provides parallel operators that enable consistent, atomic, and fault-tolerant parallel updates and queries over data stored in actors. We have used DPA to build a new query serving system, a simplified data warehouse based on the single-node database MonetDB, and enhance existing ones, such as Druid, Solr, and MongoDB, adding missing user-requested features such as load balancing and elasticity. We show that DPA can distribute a system in <1K lines of code (>10 \times less than typical implementations in current systems) while achieving state-of-the-art performance and adding rich functionality.

1 Introduction

Specialized systems that perform data-parallel, low-latency computations and frequent bulk data updates are becoming ubiquitous. These *query serving systems* include search engines like ElasticSearch and Solr [9, 13], online analytics (OLAP) systems like Druid and Clickhouse [11, 70], time-series databases like InfluxDB and OpenTSDB [14, 17], and many others [10, 15, 18, 21, 40, 51, 52]. These systems are critical to everyday applications: for example, Walmart uses ElasticSearch to check purchases for fraud in real time [6], Target and Capital One use Druid and InfluxDB for real-time monitoring in their production services [5, 7], and Facebook developed Unicorn [40] to provide graph-based search.

Developing query serving systems is challenging because their workloads typically run at large scale. Therefore, query serving system developers must implement complex distributed functionality, including data replication, update consistency, fault tolerance, and load balancing. These features vary little between query serving systems, but must be reimplemented in each of them, typically in custom distribution

layers comprising tens of thousands of lines of complex code (e.g., ~70K lines in Druid) written over many person-years. As a result of this complexity, not only are new query serving systems hard to build, but existing ones are difficult to adapt to changing user demands. For example, most query serving systems were designed for fixed-size on-premise clusters, although users increasingly deploy them in the cloud. Therefore, they do not provide user-requested cloud features such as elastic cluster auto-scaling [3, 4]. Adding any new distributed feature to an existing, large codebase can take years, even when there is strong user demand [60, 74].

Ideally, developers would be able to write query serving systems using a high-level programming model that simplifies distributing their data and computations across a cluster. Unfortunately, current distributed programming models do not support the unique workloads of query serving systems, with their combination of data-parallel low-latency queries and frequent bulk data updates. Actor models like Erlang [29], Orleans [35] and Ray [61] can manage mutable state, but lack abstractions, such as consistency and atomicity, for data-parallel operations. Parallel processing frameworks like Spark [72] can execute data-parallel queries, but lack abstractions for managing data, assuming it to be immutable.

In this paper, we propose a new programming model called *data-parallel actors (DPA)* that extends the actor model to support the unique needs of query serving systems. DPA allows developers to construct a distributed query serving system from purely single-node components, as we show in Figure 1. The DPA runtime then automatically provides the system with complex distributed features such as fault tolerance, consistency, load balancing, and elasticity.

Designing a programming model for query serving systems is challenging because of their wildly different query and data models, from search engines to timeseries databases to document stores. DPA’s insight is that the distributed functionality of a query serving system can be implemented largely independently of how the system stores and processes data on individual nodes. Therefore, DPA represents a query serving system as a collection of black-box data partitions, each encapsulated in a stateful actor. However, while conventional actor models focus on *concurrency*, where there are many actors but clients only communicate with one at a time, query serving systems also require *parallelism*: one operation can run over data in many actors, often with consistency and atomicity requirements. Thus, DPA provides parallel operators and updates over its stateful actors. Parallel operators let develop-

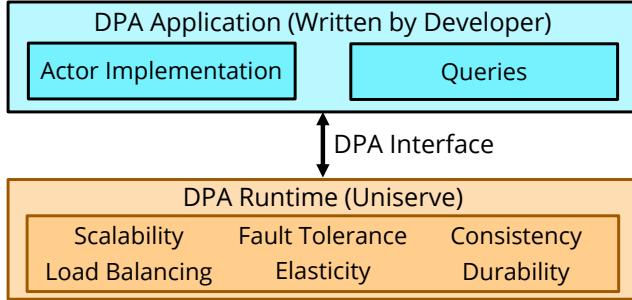


Figure 1: With DPA, a developer can construct a distributed query serving system from single-node components: code for actors and queries (blue) that implement per-node data structures and query processing logic. A DPA runtime like Uniserve (orange) manages actors and executes queries, automatically providing distributed features.

ers construct queries from generic operations such as map and broadcast, while parallel updates offer configurable consistency and atomicity guarantees. DPA defines these operations and enforces their guarantees, but is agnostic to how each node processes its part of the work. Thus, DPA *separates responsibilities* in building a query serving system, so that developers only implement single-node data structures and operations but receive a robust, performant distributed system.

We show that DPA can express the functionality of a wide range of current query serving systems, while adding powerful user-requested features that production systems lack. For example, we used DPA to wrap the existing single-node components in an OLAP system (Druid), search engine (Solr) and NoSQL database (MongoDB) into stateful actors in a few hundred lines of code. The DPA ports match the original systems on standard performance benchmarks, but also automatically receive user-demanded missing features like load balancing and elasticity, improving performance on skewed workloads by up to 3×. DPA’s generality makes it a powerful abstraction for developing new query serving systems.

We implement the DPA programming model in a runtime called *Uniserve*. Uniserve manages stateful actors and executes queries, automatically providing distributed features like durability, fault tolerance, load balancing, and elasticity. Because workloads require different implementations of these features, Uniserve allows developers to configure systems’ consistency guarantees and load balancing and auto-scaling behavior without modifying their core application code.

To evaluate DPA and Uniserve, we use them to distribute four systems: the three ports discussed above (Druid, MongoDB, and Solr) and a new simplified data warehouse we built based on the single-node database MonetDB [50]. We distribute each system with <1K lines of code. Nevertheless, on standard benchmarks, our ports match the originals’ performance, while our data warehouse matches Amazon Redshift and outperforms Spark SQL. Each DPA-based system automatically receives powerful features, including fault tolerance, durability, consistency, load balancing, and elasticity. Some

of these features, particularly load balancing and elasticity, are missing and frequently requested by users in Druid, MongoDB, and Solr. By adding these features, DPA improves these systems’ performance by up to 3× on skewed workloads. In summary, our contributions are:

- We identify *query serving systems* as an important emerging class of distributed systems defined by low-latency data-parallel queries and frequent bulk updates. We show that their workloads are not supported by existing high-level distributed programming models.
- We propose *data-parallel actors (DPA)*, a novel programming model for building distributed query serving systems from purely single-node components. We build a DPA runtime, *Uniserve*, which automatically provides fault tolerance, consistency, durability, load balancing, and elasticity to query serving systems built with DPA.
- We demonstrate the power and practicality of DPA by using it to build a simplified data warehouse and porting the popular systems Solr, Druid, and MongoDB to it. Our implementations require <1K lines of code (replacing tens of thousands) but match or outperform current systems while providing rich missing functionality.

2 Background and Motivation

In this section, we give three examples of widely used query serving systems, then make the case for DPA.

2.1 Case Studies

Apache Solr. Solr [9] is a distributed full-text search system. It provides a rich query language for searching text documents and is optimized to serve thousands of queries per second at millisecond latencies. Solr stores documents in inverted indexes based on Apache Lucene [34].

Apache Druid. Druid [70] is a high-performance analytics system. It provides fast ingestion and real-time search and aggregation of time-ordered tabular data, such as machine logs. Druid achieves its high performance through specialized *segment* data structures that store data in a tabular format optimized with summarization, compression, and custom indexes.

MongoDB. MongoDB [15] is a NoSQL document database. Unlike Solr and Druid, it is not primarily an analytics system, but is often used for analytics [16]. MongoDB performs search and aggregation queries over semi-structured data. It uses a schemaless document-oriented data format, backed up by indexes, to give users flexibility in how their data is stored and queried without sacrificing performance.

2.2 Motivating DPA

Solr, Druid, and MongoDB are popular [12] query serving systems that serve different workloads. However, while their physical data structures and query execution strategies are diverse, all use custom distribution layers to distribute data and

	Fault Tolerance	Load Balancing	Elasticity
Solr	✓	X	✓
Druid	✓	X	X
MongoDB	✓	X	X
Uniserve	✓	✓	✓

Table 1: Distributed features of query serving systems.

queries while handling failure and ensuring data consistency. These distribution layers are difficult to implement, requiring tens of thousands of lines of complex code (~90K LoC in Solr, ~70K LoC in Druid, and ~120K LoC in MongoDB).

The difficulty of distributing query serving systems complicates developing new systems, but also causes existing systems to lack user-demanded features. For example, Solr, Druid, and MongoDB struggle to provide load balancing and elasticity, as shown in Table 1. As a result, their users must over-provision clusters [45], go through the difficult and error-prone [3, 4] process of manually integrating external auto-scalers, or risk poor performance when load skews or spikes.

One reason popular systems are missing important features is that the requirements for distributed systems change over time. For example, elasticity is considered important today because most query serving systems run in the cloud, where scaling the size of a cluster is easy. However, many existing systems (including Druid, Solr, and MongoDB) were built when the cloud was less popular, so support for auto-scaling was less important and was not included. Unfortunately, the complexity of query serving systems’ distribution layers makes it difficult to add new features when users demand them. For example, adding strongly consistent replication to MongoDB required designing a novel consensus protocol because design choices made early in MongoDB’s lifetime precluded using any existing protocol [74]. Similarly, adding support for joins to Druid has been a slow, multi-year process because the system was originally built assuming queries would not require communication between data sources [60].

DPA helps solve these problems by *separating responsibilities* in a query serving system. A developer using DPA is only responsible for the core, unique functionality of their system: storing and querying data. DPA and its runtime Uniserve take responsibility for distribution and scalability, automatically providing distributed features like fault tolerance, consistency, load balancing, and elasticity. As user demands change, new features can be added to Uniserve with minimal modifications to underlying systems. This makes it easier to build new query serving systems and maintain existing ones, as they can obtain state-of-the-art distributed functionality by simply implementing the DPA interface in a ~1K LoC shim layer.

3 DPA Overview and Interface

DPA lets developers construct a distributed query serving system from purely single-node components. To use DPA, a developer must first implement an actor object that encapsu-

lates a data partition like a Solr index or Druid segment. They must then implement a query planner that translates incoming user queries to the DPA parallel operators. We show the interface for actors and operators in Figure 2.

3.1 Actors and Data

In DPA, developers express a query serving system as a collection of stateful single-node actors, each encapsulating a partition of data and exposing methods for manipulating and querying it. Query serving systems use a wide variety of data representations, from Solr inverted indexes to Druid table segments, so DPA actors can encapsulate any data structure the developer chooses for storing a collection of records. We sketch the interface for an actor in Figure 2. DPA views an actor’s implementation as a black box. Actors are only required to implement four core methods: create, destroy, serialize, and deserialize (an optional fifth method, *snapshot*, is discussed in §4.2), which the DPA runtime uses for data management. The DPA runtime also maps multiple actors to each physical machine and performs load balancing and auto-scaling. Actors typically implement other methods, e.g., custom methods for querying a search index, which are invoked by DPA operators or update functions when the runtime schedules those to run against an actor. Unlike in some general-purpose actor runtimes, actors in DPA can only communicate through DPA’s APIs; they cannot pass arbitrary messages to each other.

DPA actors are organized into *tables*, logical collections of data comprising multiple actors. Tables enable systems to manage multiple datasets and address queries and updates. To partition data across actors in a table, DPA maps records inserted in the system to actors based on *partition keys*; all records with the same key are assigned to the same actor.

3.2 Data Updates

In a conventional actor model, clients communicate with one actor at a time, updating its state directly. In a query serving system, however, users often need to update data partitioned across several actors, typically with consistency or atomicity concerns. Therefore, DPA lets developers implement parallel *update functions*, which update multiple actors in a single table. To perform updates, users implement an *UpdateFunction* interface with several methods, as shown in Figure 2.

Users invoke update functions on DPA tables and supply them with sets of records to add or change. For example, if a user is maintaining a library catalog in Solr, they might supply an update function with records containing information on new books. The runtime maps the records to actors by partition key, then runs the user’s update function on each actor with its corresponding records.

To support the diverse data models of query serving systems, DPA provides configurable consistency and atomicity guarantees for updates. These change how updates are implemented. If developers only require eventually consistent updates, they need only implement an “update” method ap-

Actor Interface	
<i>create()</i> → Actor	Create a new (empty) actor.
<i>destroy()</i>	Destroy an actor.
<i>serialize()</i> → File	Serialize an actor's data to files on disk.
<i>deserialize(File)</i> → Actor	Reconstruct an actor from files on disk.
<i>snapshot()</i> → Actor	Create a snapshot of an actor's data.
Record Interface	
<i>getPartitionKey()</i> → Int	Get a record's partition key.
Update Function Interface	
<i>updatedTableName()</i> → Table	Name of the table to be updated.
<i>consistencyLevel()</i> → Level	What consistency level to use? (§4.2)
<i>update(Actor, List[Record])</i>	Apply eventually consistent update to an actor.
<i>prepare(Actor, List[Record])</i> → Bool	Prepare a serializable update.
<i>commit(Actor)</i>	Atomically make prepared changes visible.
<i>abort(Actor)</i>	Roll back prepared changes.
Parallel Operator Interface	
<i>inputs()</i> → List[Operator Table]	What are the input operators and tables?
<i>keysToQuery()</i> → Map[Int, List[Int]]	For inputs, what partition keys are used?
<i>operator()</i> → OperatorFunction	The operator function. Signature depends on the operator (see below).
Parallel Operator Functions	
<i>map(Actor) → Data</i>	Apply a transformation to data.
<i>scatter(Actor) → List[(K, C)]</i>	Partition data into (attribute, chunk) pairs.
<i>gather(K, List[C], Actor) → Data</i>	Combine chunks with the same attribute, plus actors whose partition key matches that attribute; materialize the output.
<i>query(Actor) → V</i>	Query an actor to obtain a value.
<i>combine(List[V]) → V'</i>	Combine values into a query answer.

Figure 2: The DPA interface. It consists of callback functions implemented by the developer and invoked by Uniserve to manage data and execute queries.

plying an update to an actor. However, if they need stronger guarantees such as serializability, they must implement the participant protocol of two-phase commit (prepare, commit and abort). We discuss consistency in Section 4.2.

3.3 Queries

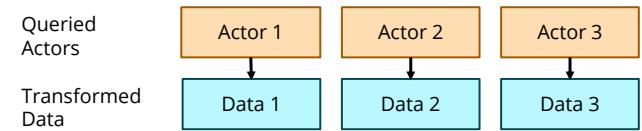
Unlike traditional actor models, query serving systems execute parallel queries over data stored in many actors. To enable these queries, DPA provides a small but general set of *parallel operators* which let developers construct queries from generic operations like map and broadcast. We list the parallel operators in Figure 2 and diagram them in Figure 3.

Users write queries by subclassing one of several parallel operator classes (e.g., MapOperator) and implementing appropriate callback functions. Queries may be composed of multiple operators. In practice, we expect developers to implement a query planner in their system's client library that translates queries to DPA operators for execution by the DPA runtime. Most existing query serving systems have similar planners. Both the query planning logic and operator execution callbacks can be single-node: the DPA runtime handles the work of distributing a plan's computation by executing each operator on each actor that contains relevant data.

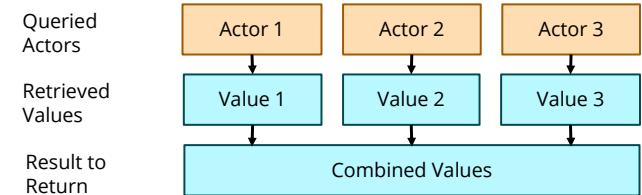
Operators cannot modify actor state (only updates can), but instead materialize output data that later operators can read. The input to each operator is a list of tables and of data materialized by other operators. Operators can specify what partitions of their input data to query through their *keysToQuery* method, listing specific partition keys for each input.

DPA provides five generic parallel operators that we found sufficient to support the serving systems we considered (Sec-

a) **Map Operators** apply a transformation to several actors in parallel, materializing the transformed data.



b) **Retrieve and Combine Operators** compute the result of a query. A retrieve operation computes values from actors in parallel, then a combine operation combines them into a query result.



c) **Scatter and Gather Operators** enable collective operations. A gather operation computes (attribute, data chunk) pairs from actors. A scatter operation combines chunks with the same attribute and materializes the result. Scatter can also (not shown) combine chunks with other actors whose partition key matches the chunk attribute.

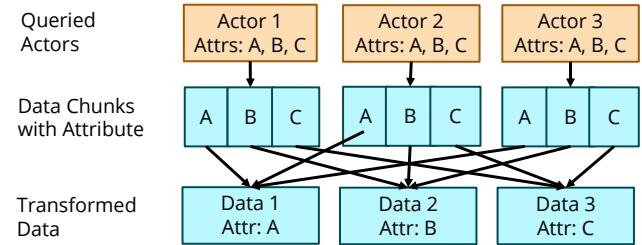


Figure 3: The five DPA parallel operators.

tion 5), though more operators could be added:

Map. The *map* operator applies a function to actors in parallel and materializes the transformed data. For example, a map operator might search for documents in a collection based on a field, or in a subset of actors specified via *keysToQuery*.

Retrieve and Combine. The *retrieve* operator computes a value from an actor and returns it to the DPA client. It is used to retrieve the results of a query. If retrieve is executed on many actors in parallel, it must be followed by a *combine* operator, which aggregates retrieved values. Retrieve and combine must be the last two operators executed in a query. For example, if in Solr we have several actors storing indexed text data and wish to search it for the word “computer,” we can execute a retrieve operation to find documents containing the word “computer” on each actor, then combine these results.

Scatter and Gather. The last two operators, *scatter* and *gather*, provide data communication between actors, enabling collective operations such as broadcast and shuffle. The scatter operator produces from an actor a set of (attribute, chunk) pairs, where the attribute can be any value and the chunk contains data stored in a developer-defined serialized format. A

scatter operator must be followed by a gather operator. Gather executes one time for each attribute produced by the preceding scatter. Each execution of gather takes in all data chunks associated with that attribute, along with any actor containing data whose partition key matches the gather attribute, and materializes combined and transformed data.

To demonstrate scatter and gather, consider a shuffle join in a data warehouse setting. Say we have tables of customer data $C(c_id, country)$ and order data $O(o_id, c_id, price)$, both partitioned across several actors. We wish to compute the total amount of money spent by each French customer: `SELECT c_id, SUM(price) FROM C, O WHERE C.country='France' GROUP BY c_id`. First, we execute a scatter operation on every actor containing data from C or O . This operator returns (attribute, chunk) pairs where every attribute corresponds to a set of customers (range of values of c_id) and every chunk contains data associated with those customers. We then execute a gather operator on the results of the scatter. Each gather execution takes in a unique attribute and all its associated chunks. In other words, it takes in all records from both tables corresponding to a set of customers. The operator executes the original query on this data, computing the amount of money spent by each French customer. Each execution materializes new data containing results for a different set of customers; this data collectively forms the result of the original query. Subsequent operators could then query this data; for example retrieve and combine operators could be used to find the ten top-spending French customers.

3.4 Case Study: Solr

We now describe how to create a DPA port of the distributed full-text search system Solr [9]. Natively, Solr stores data by sharding text documents across Lucene inverted indexes (custom data structures enabling ultra-fast search [34]) on several machines. When Solr receives a new document, it hashes it, uses the hash to pick a shard, and adds it to that shard’s index. To port Solr’s distributed data storage capabilities to DPA, we encapsulate inverted indexes in actors. We add data to actors in units of Solr documents, which act as DPA records. Just like Solr, we hash documents to obtain a partition key, then use it to assign them to actors.

All Solr queries are searches: they take in a criterion, such as a query string, and return a list of documents that satisfy it. This list may be aggregated by grouping or faceting. Natively, Solr distributes queries by searching each shard separately, then combining results on a single node [20]. To port Solr’s distributed query capabilities to DPA, we must translate Solr queries to DPA queries. Because all Solr queries are searches, we can implement them using DPA retrieve and combine operators. Each retrieve operator searches its target actor for a set of results, then the results are combined and returned. For example, in a query that searches for books whose title contains the word “goblin,” retrieve operators run in parallel on every queried actor, searching their data for “goblin.” A

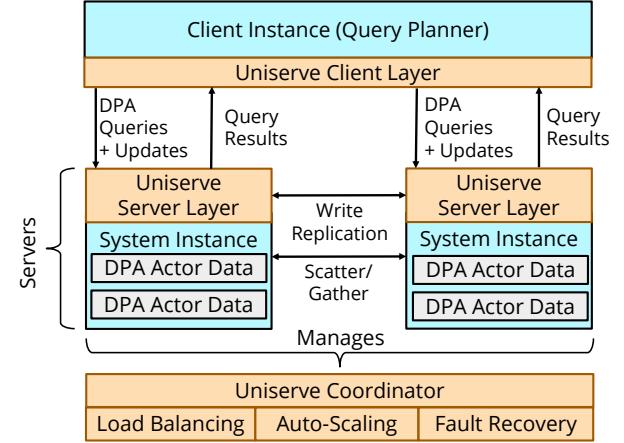


Figure 4: The Uniserve architecture. Servers run a thin Uniserve layer (orange) above actors encapsulating partitions of data (gray) stored in instances of the underlying system (blue). A coordinator manages cluster state and provides distributed features.

combine operator then combines the results. The DPA port of Solr is implemented in <1K lines of code (replacing ~90K lines of native Solr code), and can execute any query recognized by the standard Solr parser. As we show in Section 7, our port matches native Solr performance while providing features lacking in native Solr, such as load balancing.

4 Uniserve: A Runtime for DPA

We implement DPA in a runtime called Uniserve. In the DPA programming model, developers take responsibility for implementing actors and queries on a single node, but Uniserve takes responsibility for distributing them, managing actors and executing queries at scale. Uniserve automatically provides critical distributed features such as fault tolerance, durability, consistency, load balancing, and elasticity.

4.1 Architecture

A Uniserve cluster consists of many data servers. Each runs a thin Uniserve layer over developer-provided single-node code responsible for physical data storage. Clients send queries and updates to servers. Each client runs a thin Uniserve layer above a developer-provided query planner. A central coordinator manages cluster state with the help of ZooKeeper [49]. Uniserve additionally requires an external durable storage system (e.g. S3 or HDFS) to back up data. We diagram the Uniserve cluster architecture in Figure 4.

Servers store data and execute queries. In each server, a thin Uniserve layer runs above developer-provided single-node code responsible for physical data storage and query execution. For example, if we were to distribute Solr using DPA and Uniserve, each server would run a Uniserve layer above a single-node Solr instance. DPA actors encapsulate physical partitions of data stored in this system, so for example each actor might encapsulate a Solr inverted index. The Uniserve layer facilitates query execution. It receives query operators

and updates from clients and executes them in the underlying system using the DPA interface. Additionally, it handles update replication, maintains a log of the most recent updates, periodically backs up data to durable storage, and transfers actors between servers in response to coordinator commands.

Clients plan queries and submit them to servers. In each client, a thin Uniserve layer runs alongside a developer-provided query planner. The query planner receives user queries (or update requests) in some query language and translates them to DPA parallel operators (or update functions). The client then submits these to the appropriate servers, eventually receiving and returning a result. Clients learn actor locations from the coordinator and ZooKeeper so they know to which servers to send queries or updates.

A Uniserve cluster contains a single coordinator that manages cluster state. It is responsible for many distributed capabilities including load balancing, failure recovery, and elasticity, which we discuss in more detail later. It backs up cluster state to ZooKeeper. To minimize query latency at scale, the coordinator is entirely off the query critical path.

4.2 Update Consistency and Atomicity

Conventional actor runtimes do not provide cross-actor data consistency guarantees, assuming operations occur on a single actor at a time. Query serving systems, however, perform parallel updates on partitioned and replicated data, so Uniserve provides cross-actor consistency and atomicity guarantees.

Query serving systems typically ingest bulk data for analytics; for example time series in Druid or logs in Solr. Updates are usually append-only, but modification of existing data is possible. Most updates are batched, and systems provide high update throughput but not necessarily low update latency. Many systems, like Druid, do *not* support transactional semantics. However, they still provide update consistency and atomicity guarantees of varying strength.

Uniserve automatically provides primary-backup actor replication and data consistency guarantees to query serving systems. Because query serving system data models vary, we make these guarantees *configurable*: when implementing an update function (§3.2), developers can choose a level of consistency appropriate to their data model. Uniserve provides the consistency levels most common in the query serving systems we surveyed. In the remainder of this section, we describe these guarantees and what developers must implement to obtain them. Then, in Section 4.3, we explain how Uniserve upholds its guarantees in case of failures.

Eventual Consistency. By default, Uniserve provides eventual consistency, guaranteeing only that all replicas of an actor eventually converge to the same state. Many systems, like Solr and Druid, use eventual consistency [19]. To write an eventually consistent update function, developers need only implement an “update” method. To execute an eventually consistent update, Uniserve applies it to the primary of an actor synchronously, then replicates it asynchronously. All replicas

of an actor apply the same updates in the same order.

Serializable Updates. Uniserve can guarantee serializability for updates, so the outcome of a sequence of updates is equivalent to the outcome of the updates executed serially. As implemented, this also guarantees linearizability, so read queries made after an update completes always reflect the update. This functionality was recently added to MongoDB [74] and is common in data warehouses. To write a serializable update function, developers must implement the participant protocol of two-phase commit, with separate prepare, commit, and abort stages. Uniserve only commits an update if it has successfully prepared on all actors and their replicas, aborting if failures occur. We currently do not allow multiple serializable updates to run concurrently on the same table, but plan to add concurrency control in the future.

Full Serializability. Uniserve can make updates serializable (and therefore atomic) with respect to read queries, so a parallel read query either sees an update applied to all actors or to none of them, as in SQL databases. To obtain this guarantee, developers must both provide serializable update functions and implement the optional *snapshot* actor method (Figure 2). Using this method, Uniserve creates a versioned copy of each actor’s data upon update and ensures that read queries see consistent data versions across actors. We expect developers to implement snapshot using optimizations such as shadow paging and copy-on-write to minimize its cost.

4.3 Fault Tolerance and Failure Recovery

Uniserve assumes a fail-stop model for failures, where the only way servers fail is by crashing. It also assumes that if a server crashes, it remains crashed until restarting (when it will be treated as a new server). Moreover, it assumes the coordinator and ZooKeeper are always available; if either fails the cluster will be unavailable until they are restarted, with the coordinator restoring its state from ZooKeeper.

Durability. Uniserve provides update durability through replication and through asynchronous backup to durable storage such as S3. If all replicas of an actor fail, the coordinator orders a random surviving server to load the actor from durable storage. Thus, Uniserve can only lose data if all replicas of an actor fail, and will only lose data committed since the last backup. Additionally, eventually consistent updates can be lost if the primary fails before the updates are replicated.

Update Fault Tolerance. When providing eventual consistency, Uniserve only guarantees that all replicas of an actor will eventually converge to the same state. Therefore, it is possible for an update to partially succeed—to succeed on some actors but fail on others. If the primary of an actor fails, the coordinator chooses the replica with the most advanced update as the new primary, relying on the guarantee that all replicas apply the same updates in the same order. All other replicas then sync with the new primary, applying missing updates from its log to converge to its state.

When providing serializability, Uniserve guarantees that all updates either totally succeed or abort. Uniserve only commits an update if it has successfully prepared on all actors and their replicas, aborting if any failures occur. To ensure the cluster remains in a consistent state in case of a client crash, the client writes ahead any commit or abort decision to ZooKeeper; servers can reference this if the client fails (or abort if the client fails before making a decision).

Query Fault Tolerance. If a failure occurs during the execution of a parallel operator on an actor, the client retries with a different replica. It keeps retrying until it has exhausted all replicas; this occurs only if all are lost, in which case the actor must be restored from durable storage and the query fails.

4.4 Load Balancing and Data Placement

Query serving systems often have unpredictable workloads skewed towards a small number of data items or partitions, so load balancing is necessary for consistent performance [45]. The obvious way to balance load is through fine-grained query scheduling, but this is impractical for query serving systems because of their strict latency requirements and because all queries must run on specific data partitions. Instead, Uniserve balances load through data placement, managing the actor-to-server assignment to ensure no server is overloaded.

By default, Uniserve provides a greedy load balancing algorithm, similar to that of E-Store [67], which repeatedly moves the most-loaded actors from the most-loaded servers to the least-loaded servers while also replicating actors whose load exceeds average server load. However, some applications may want to instead use a custom algorithm. Therefore, we allow developers to define a *data placement policy*, which uses information on cluster utilization to compute an assignment of actors to servers. If a policy is provided, Uniserve takes responsibility for collecting its input data and implementing its output assignment, moving actors to new locations.

A data placement policy must be expressed as a function that takes in the total query load (self-reported by the underlying system) and memory and disk usage of each actor, as well as the current assignment of actors to servers. It returns an updated assignment of actors to servers, expressed as a map from actor number to a list of server IDs. Assignments may replicate actors across multiple servers, either for redundancy or to spread out their load.

To physically move actors during load balancing, Uniserve first prefetches, from durable storage, replicas of reassigned actors on their target servers. These then sync with the actor primary, applying updates from its log. Only after replicas are ready does Uniserve notify clients of the actor movement. Then, after notifying clients, it deletes the original copies of the actors if necessary. If some of the deleted actors were primaries, Uniserve designates randomly selected replicas as new primaries. This procedure ensures high query availability during shard transfer, but if a primary is removed updates may briefly block while a new primary is designated.

	System Type	Data Type	Query Operations
Druid [70]	OLAP	Indexed Tables	Aggregations, joins
Pinot [51]	OLAP	Indexed Tables	Aggregations
ClickHouse [11]	OLAP	Indexed Tables	Aggregations, joins
Atlas [10]	Timeseries DB	Time series	Aggregations
InfluxDB [14]	Timeseries DB	Time series	Aggregations
Solr [9]	Full-Text Search	Indexed text	Text search
ElasticSearch [13]	Full-Text Search	Indexed text	Text search
Unicorn [40]	Graph Database	Social Graphs	Graph Search
FAISS [52]	Vector Database	Vectors	Vector Search
Pinecone [18]	Vector Database	Vectors	Vector Search
Vespa [21]	Vector Database	Vectors	Vector Search
MongoDB [15]	NoSQL	Documents	Aggregations, search
MonetDB [50]	Data Warehouse	Relational Tables	SQL

Table 2: Systems we believe can be distributed with or ported to DPA and their properties. Systems we have implemented are in bold.

4.5 Elasticity and Auto-Scaling

Query serving system load often varies over time, so they benefit from *elasticity*, the ability to dynamically adjust cluster size. As a result, when deployed in an elastic cloud environment such as EC2, Uniserve automatically scales cluster size in response to load changes.

By default, Uniserve provides a utilization-based auto-scaling algorithm similar to the algorithms used in cloud auto-scalers [32]. It adds servers if CPU utilization exceeds an upper threshold and removes them if it is below a lower threshold. However, like in load balancing, Uniserve also gives developers the option of defining their own *auto-scaling policy*, which uses information on cluster utilization to decide whether to add or remove nodes. Uniserve provides the policy with its input and physically executes its commands, adding or removing nodes and transferring actors as necessary.

An auto-scaling policy must be defined as a function that takes in the CPU utilization, memory and disk usage, and total query load of each server. It returns the number of servers to be added or removed, as well as the IDs of the servers to be removed, if any (chosen randomly if there is no preference).

Uniserve periodically executes the policy (using a configurable interval) and adjusts cluster size. After adding or removing a server, Uniserve uses the load balancer to reassign actors; if servers are removed this reassignment is done preemptively so availability is not affected.

5 Generality of DPA

In this section, we demonstrate the generality of DPA by describing some of the diverse systems it can distribute, summarized in Table 2. We also discuss its limitations.

OLAP Systems and Time Series Databases. OLAP systems rapidly answer multidimensional analytics queries over tables. They are closely related to time series databases, which query time-ordered data. Both typically store data in a compressed and indexed columnar format. Their workloads usually filter, group, and aggregate this data. This naturally fits DPA: we partition data by key columns across actors (e.g., by time range) to support partition filtering, and implement

most aggregations with retrieve and combine operators, using scatter and gather to shuffle or broadcast data if necessary. We implement a port of Druid [70], which is both an OLAP system and timeseries database, on DPA; its design patterns generalize to others from both categories such as Pinot [51], Clickhouse [11], Atlas [10] and InfluxDB [14].

Full-Text Search. Full-text search systems execute search queries over text data stored in specialized data structures such as inverted indexes [34]. Because all their queries are searches, they are easy to fit to DPA, as we showed in Section 3.4. We implement a port of one full-text search system, Solr [9], and can generalize to others like ElasticSearch [13].

Vector Databases. Vector databases store data using vector indexes to perform fast nearest neighbor search, often for machine learning workloads. Recent examples are Pinecone [18], Vespa [21], and FAISS [52]. Like full-text search systems, they easily fit DPA as their queries are searches.

Graph Databases. Graph databases represent data using a graph data model. Some graph database queries are data-parallel, including whole-graph algorithms like PageRank and queries like finding all checkins at a certain location in a social network graph [40]. Others are not; for example, a graph traversal query, like finding all nodes within N hops of a target, is most efficiently implemented using breadth-first search, not data-parallel operators such as iterative self-joins. Data-parallel graph databases such as Facebook’s Unicorn [40] search engine fit the DPA programming model.

Other Systems. DPA can distribute other systems with data-parallel queries. For example, we implement a DPA port of the NoSQL document store MongoDB. We also implement a simplified OLAP data warehouse based on the single-node columnar database MonetDB.

Limitations of DPA. DPA has two major limitations. First, its query model works best for data-parallel queries. As we have shown, this is sufficient for many popular query serving systems, but not some specialized query types like graph traversal queries. Nonetheless, we believe DPA would have made many of today’s query serving systems easier to develop, and can augment them with missing functionality.

Second, DPA is not designed to provide low latency for small point updates, especially with transactional guarantees. Small transactional updates are rare in query serving systems because these are often updated in bulk (e.g., using data collected in a message queue like Kafka). However, they are common in other contexts such as online transactional processing (OLTP) workloads, which DPA does not target.

6 Distributing Systems with DPA

To demonstrate the practicality of DPA, we use it to distribute four systems. First, we port Druid, Solr, and MongoDB to DPA, replacing their native distribution layers. Then, we build

a new system using DPA: a simplified data warehouse based on the single-node column store MonetDB.

We implement each of our four systems in <1K lines of code (LoC). This number includes all code needed to implement the DPA interfaces with each system’s already-existing single-node implementation, but not any code in Uniserve. This demonstrates that DPA simplifies building distributed query serving systems, as it replaces custom distribution layers totaling ~90K LoC in Solr, ~120K LoC in MongoDB, and ~70K LoC in Druid. For comparison, Uniserve itself is ~10K LoC. This smaller size is because Uniserve makes use of tools like ZooKeeper and gRPC for basic functionality that other systems implemented themselves.

Solr. We described the port of Solr in Section 3.4.

Druid. In our port of Druid [70], actors encapsulate single-node Druid datasources. These are analogous to database tables and are backed by Druid segments, which are optimized tabular stores for timeseries data. We implement most actor manipulation and update functionality using the Druid datasource API. Serializing and deserializing data is easy because Druid segments live in portable directories on disk.

All Druid queries aggregate filtered and grouped data from datasources. Our port supports most common Druid queries: simple aggregations (sums, counts, or averages) of filtered and grouped data. It could easily be extended to support any other query by adding support for more aggregation operators. Our Druid queries use retrieve and combine operators to separately query actors then aggregate the results. Druid uses a similar model natively. We can also use scatter and gather operators to support Druid’s recently-added [60] broadcast joins.

MongoDB. In our port of MongoDB [15], actors encapsulate single-node MongoDB collections, analogous to database tables. We implement most actor manipulation and update functionality using the MongoDB API for manipulating collections. We implement actor data serialization and deserialization using the mongodump and mongorestore tools.

MongoDB queries apply an “aggregation pipeline” of operators to a collection. These operators perform tasks such as filtering, grouping, and accumulating documents. We can support any MongoDB operator, but so far have only implemented operations for filtering, projecting, summing, counting, and grouping data. Our query implementations are similar to those in our Druid port and those in native MongoDB: querying actors separately, then combining the results.

MonetDB. We have built using DPA a simplified data warehouse based on the single-node column store MonetDB [50]. It stores data in MonetDBLite [65], the embedded implementation of MonetDB. Each server runs MonetDBLite embedded in the same JVM as the Uniserve layer. Actors encapsulate MonetDB tables and implement interface methods using equivalents in the MonetDBLite API.

Our simplified data warehouse supports a large subset of

SQL, including selection, projection, equijoins, grouping, and aggregation. We implement simple aggregation queries with retrieve and combine operators, as in other systems. To execute more complex queries, such as joins, we use scatter and gather operators to shuffle or broadcast data, then use retrieve and combine operators to produce a query result.

7 Experimental Evaluation

We evaluate DPA and Uniserve using the four systems discussed in Section 6. As we have shown, DPA makes distributing these systems considerably simpler; each requires <1K lines of code to distribute as compared to the tens of thousands of lines in custom distribution layers (~90K in Solr, ~120K in MongoDB, and ~70K in Druid). Our evaluation shows that:

1. Distributed systems built using DPA and a specialized single-node system, such as our MonetDB-based simplified data warehouse, can match or outperform comparable distributed systems such as Spark-SQL and Redshift.
2. DPA ports of distributed systems match the performance of natively distributed systems under ideal conditions, such as static workloads without load skew.
3. DPA ports of distributed systems provide new features such as elasticity and load balancing and so outperform natively distributed systems under less ideal conditions – workloads that change, have load skew, or have failures.

7.1 Experimental Setup

We run most benchmarks on a cluster of m5d.xlarge AWS instances, each with four CPUs, 16 GB of RAM, and an attached SSD. We evaluate using Apache Solr 8.6.1, Apache Druid 0.20.1, MongoDB 4.2.3, and MonetDBLite-Java 2.39. We use four data servers for smaller-scale benchmarks and forty for large-scale benchmarks. In both cases, an additional node is set aside for the coordinator.

When benchmarking Solr, Druid, and MongoDB natively, we place the master (Solr ZooKeeper instance, Druid coordinator, MongoDB config and mongos servers) on a machine by itself and a data server (SolrCloud node, Druid historical, MongoDB server) on each other node. We also disable query caching and set the minimum replication factor to 1.

When benchmarking systems with Uniserve, we use the implementations described in Section 6. We place the Uniserve coordinator and a ZooKeeper server on a machine by themselves and data servers on the other nodes.

7.2 Experiment Workloads

We evaluate each system with a representative workload taken when possible from the system’s own benchmarks. All of our comparison systems achieve state-of-the-art performance on their benchmarks, so DPA also achieves state-of-the-art performance by matching them. We benchmark Solr with queries from the Lucene nightly benchmarks [59]. We run each query

on a dataset of 1M Wikipedia documents (more for large-scale benchmarks) taken from the nightly benchmarks. We use two representative nightly benchmark queries—an exact query for the number of documents that include the phrase “is also” and a sloppy query for the number of documents that include a phrase within edit distance four of the phrase “of the.”

We benchmark Druid with two of the benchmark queries from the Druid paper [54, 70]. These are TPC-H queries modified by the Druid developers to reflect the strengths of Druid; we run each against 6M rows of TPC-H data. The queries we use are `sum_all`, which sums four columns of data; and `parts_details`, which performs a group-and-aggregate.

We benchmark MongoDB using YCSB [39], simulating an analytics workload. Before running the workload, we insert 10M sequential items (10GB of data) into the database. We run a workload of 100% scans, where each scan retrieves one field from each of uniformly between 1000 and 2000 items. We base our YCSB client implementation on the MongoDB YCSB client from the YCSB GitHub repository [22].

We benchmark our data warehouse using representative TPC-H queries (Q1, Q3, and Q10) at scale factors of 5 and 25, requiring 5GB and 25GB of data respectively.

7.3 Benchmarks

Ideal Conditions. We first benchmark our Solr, Druid, and MongoDB ports on a uniform workload where each data item is equally likely to be queried. We run each benchmark with several client workers; each repeatedly makes the query and waits for it to complete, recording throughput and latency. We start with a single worker and add more until throughput no longer increases, showing results in Figure 5. We find that, as expected, our ports’ performance is similar to native system performance on all benchmarks.

Scalability. We next evaluate Uniserve scalability, scaling the Solr benchmarks with one client worker from four to forty servers. We scale the amount of data to maintain a constant 5 GB of data per server. We show results in Figure 6. Because all queries access all data, we expect performance to be near-constant as the number of servers (and amount of data) increases, and indeed it is.

Data Warehouse Benchmarks. We next benchmark our simplified data warehouse based on MonetDB, comparing its performance with native MonetDB, Spark-SQL [28], and Redshift [47]. We use three TPC-H queries: Q1, an aggregation query; Q3, a three-way join; and Q10, a four-way join. We implement Q3 and Q10 using scatter and gather operators to perform both broadcast and shuffle joins. We show results in Figure 7. We run multiple trials of each benchmark, reporting the average of results after performance stabilizes. This ensures Spark-SQL and Redshift can cache data in memory.

We first investigate the overhead Uniserve adds to single-node MonetDB. On a single node, our data warehouse performs the same as native MonetDB on the aggregation query

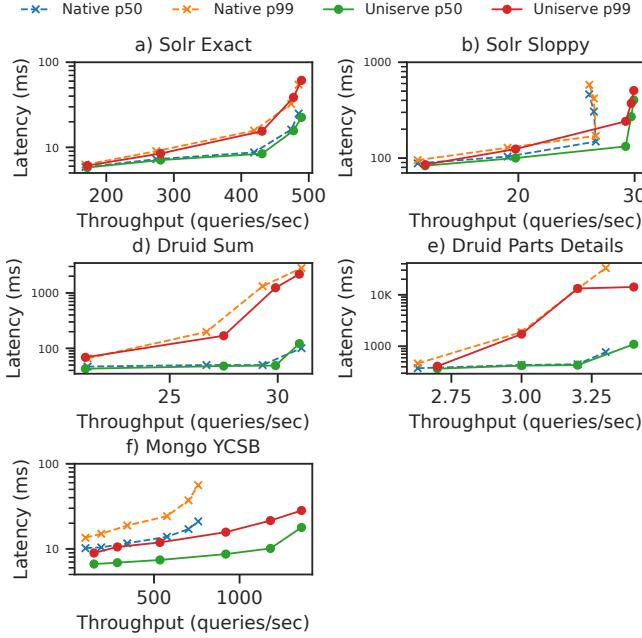


Figure 5: Throughput versus latency for native systems and DPA ports on uniform and static query workloads. Our ports match native system performance.

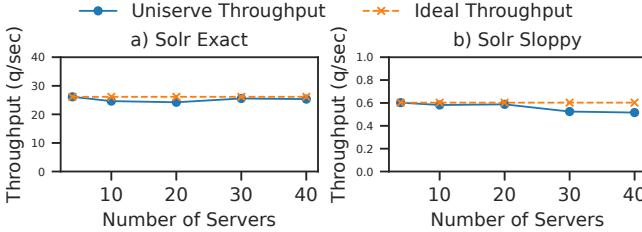


Figure 6: Uniserve scalability on the Solr benchmarks.

Q1 but worse on Q3 and Q10 due to the communication cost of shuffling. We then compare our system to Spark-SQL and Redshift on 160 cores (forty servers for Uniserve and Spark-SQL, five dc2.8xlarge Redshift servers). We find that our data warehouse outperforms Spark-SQL and matches Redshift. This shows that by distributing a single-node system like MonetDB, DPA can in <1K lines of code match or outperform popular distributed systems like Redshift and Spark-SQL on their core workloads.

Update Performance. We next investigate Uniserve update performance. We benchmark 1 MB, 10 MB, and 100 MB updates on Solr, Druid, and MongoDB, using each system’s benchmark dataset. We use these bulk writes because they are typical of query serving system workloads. We compare native system performance to Uniserve performance, showing results in Figure 8. For Solr and Druid, we provide eventual consistency, matching those systems’ semantics; for MongoDB we enable update serializability (through two-phase commit in Uniserve) and perform the update on four partitions in parallel. We find that across the board, Uniserve matches

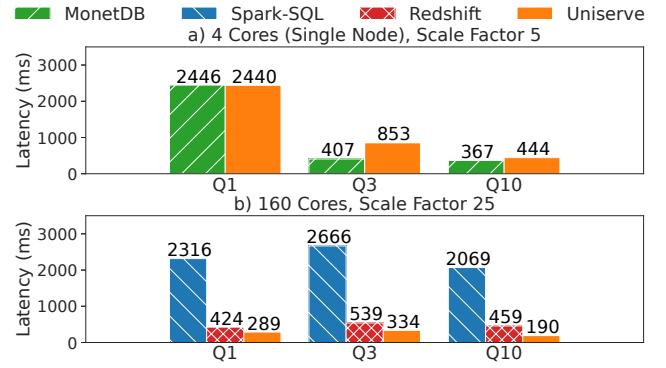


Figure 7: Comparison between our simplified data warehouse, single-node MonetDB, Spark-SQL, and Redshift on TPC-H queries Q1, Q3, and Q10 on 4 cores (single-node) and on 160 cores with TPC-H scale factors of 5 and 25. Uniserve is competitive on a single node and outperforms Spark-SQL and matches Redshift at scale.

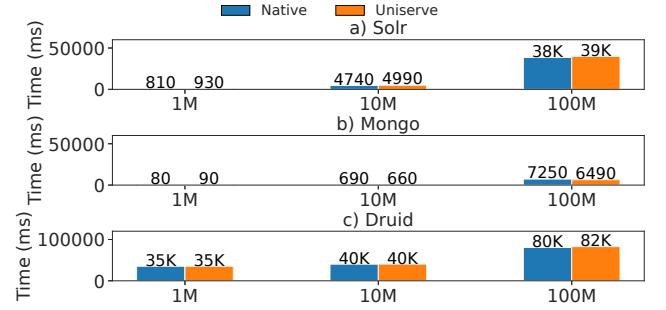


Figure 8: Execution time of 1 MB, 10 MB, and 100 MB updates with native systems and with Uniserve. Uniserve matches native system performance.

native system update performance.

Hotspots. To demonstrate the importance of load balancing, we next investigate the performance of the default Uniserve load balancer on benchmarks with load skew. We compare against Druid, whose load balancer ensures each server hosts the same amount of data but does not balance query load. First, we execute a workload where 7/8 of the queries are sent to a single slice of data (four months) and scatter the rest uniformly on the remainder of the data, showing results in Figure 9a. Because Uniserve balances load in the hotspot, it outperforms Druid by up to 3×.

We next repeat the experiment, fixing the number of clients at twelve but varying the fraction of queries sent to the hotspot. We show results in Figure 9b. We find that changing skew does not affect Uniserve performance because Uniserve keeps load balanced under any load distribution. However, Druid performance worsens with increasing skew.

Dynamic Load. To demonstrate the importance of elasticity in query serving systems, we next investigate the performance of the default Uniserve auto-scaler on a dynamic workload. We run the Solr sloppy benchmark for six hours sending

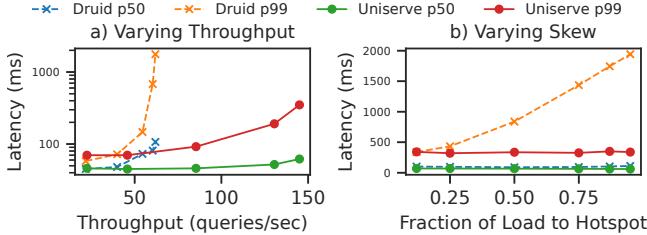


Figure 9: Effect of query skew and load balancing for Druid- and Uniserve-distributed queries. On the left, we vary throughput in a workload where one slice of data receives $7/8$ of queries; Uniserve balances load and so outperforms Druid. On the right, we vary the fraction of queries received by the hot slice; Uniserve keeps performance constant as skew increases but Druid does not.

queries at a target throughput, which varies from 240 to 1300 uniformly distributed queries per minute. Uniserve starts with one server and adds or removes more as load changes. We show results in Figure 10. We see that Uniserve is always able to scale to meet the target throughput. As load increases, it adds servers so there are always enough to process each query in time. As load decreases, it removes unnecessary servers but keeps enough to process incoming queries. Because the target query runs in parallel on all actors, adding servers decreases latency (as the query can run in parallel on more cores on more servers) and removing servers increases latency.

Importantly, Uniserve can resize clusters without losing performance. By prefetching replicas of moved actors onto new servers before serving any queries, Uniserve guarantees that queries need not contend with actor transfers for resources. As a result, Uniserve can add or remove servers without affecting throughput or median latency. Tail latency does spike briefly when a server is added, but this represents only the handful of queries sent between when Uniserve notifies servers of the new server and when it notifies clients.

Failures. We next investigate how Uniserve deals with server failures, using the Druid sum_all benchmark. We run this benchmark for ten minutes with a client sending 500 asynchronous queries uniformly per minute. Three minutes into the benchmark, we kill -9 a data server. We record how many queries succeed during each minute of the benchmark. We run the benchmark twice, once starting with four replicas of each data partition or actor (one on each server), and once with just a single replica. We show results in Figure 11.

When all servers have replicas of all partitions (11b), Uniserve recovers instantly, routing queries to replicas. Druid, however, takes thirty seconds to route queries to replicas, resulting in hundreds of query failures. When there is only one replica of each partition (11a), both systems fail hundreds of queries but recover within thirty seconds by restoring replicas from durable cloud storage. However, while all queries sent to Uniserve either fail or successfully complete, some “successful” Druid queries return incorrect results. This experiment confirms previously-reported issues Druid faces in large-scale

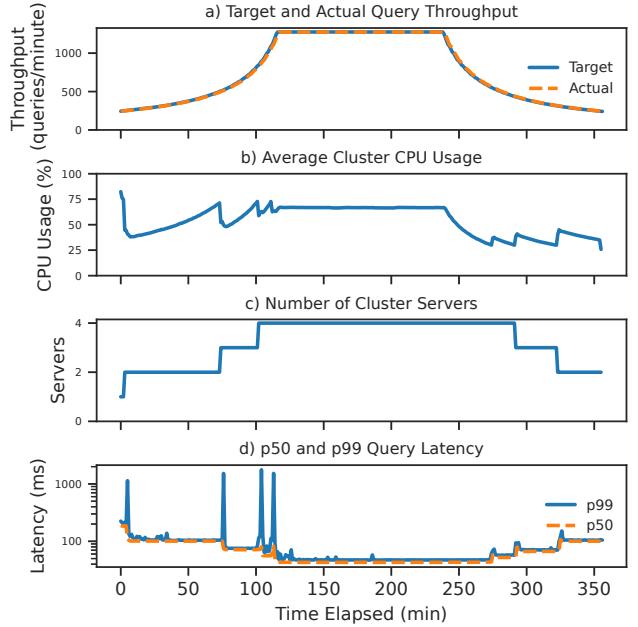


Figure 10: On the Solr sloppy benchmark with Uniserve auto-scaling, varying target throughput and observing effects on actual throughput, average cluster CPU usage, the number of cluster servers, and query latencies. Uniserve scales the cluster so that actual throughput always matches target throughput; resizing causes only brief (<1 sec) spikes in query latency. Latency decreases as cluster size increases because all queries run on all data and their parallelism increases as the data is spread over more servers.

deployments [56] and shows Uniserve can address them.

8 Related Work

Actors Actor models are abstractions for concurrent computation built around stateful agents called actors [26]. Prior surveys [53] identified five characteristics of an actor model: actors encapsulate their own state, exhibit location transparency, are mobile, are scheduled fairly, and communicate through message passing. DPA actors exhibit four of these properties: they encapsulate shards of data, are addressable through partition keys, can be moved between servers, and share resources on each machine. However, unlike prior actor models, DPA actors do not communicate via message passing but are instead acted on by parallel operators and updates. Other systems based on actors include Erlang [30], a programming language with built-in actor support; Akka [8], which supports actors on the JVM, including persistent actors with durable state; Orleans [33,35], which supports virtual actors that are only instantiated on-demand when required; and Ray [61,69], where developers can call remote procedures on stateful actors.

Critically, most existing actor models focus on *concurrent* computations, not the *parallel* ones performed by query serving systems and DPA. Most actor models do not support cross-actor transactions, requiring users to manually implement protocols such as two-phase commit. Even in systems

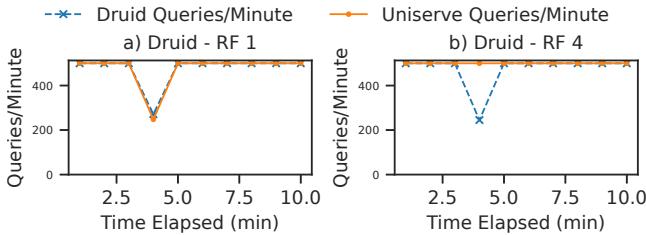


Figure 11: Query throughput (targeting 500 queries/min) of Druid- and Uniserve-distributed `sum_all` queries when one data server is killed after three minutes. The left graph shows performance starting with a single replica of each partition; the right graph with four.

like Ray which allow many actors to be accessed in parallel, there is only limited support for collective operators like gather or scatter [23] and no support for consistency or atomicity guarantees across actors. DPA instead reasons about parallel operators directly, taking advantage of the fact that query serving systems mostly need to support bulk updates as opposed to many concurrent write transactions, and offers multiple consistency levels to support different system designs. Eldeeb and Bernstein extended Orleans with a transactional actor concept [43], but that work focused on allowing clients to make multiple calls to the same actor as a single transaction and tracking these calls’ effects on downstream actors through message passing, which would be expensive for the large data-parallel operations that DPA targets. Early versions of Akka also supported transactional actors on the same server [2], but this was removed because the mechanism was hard to extend to multiple servers [1].

Other Distributed Programming Models One class of programming model often used for parallel queries are batch frameworks like MapReduce [42], Hadoop [66], Percolator [64], Dryad [71], and Spark [72]. Unlike query serving systems, these only execute computations and do not provide abstractions for managing data, typically assuming its immutability. Moreover, they are not designed for low latency and typically do not implement many of the optimizations used in query serving systems, such as augmenting data with secondary indexes. Researchers have attempted to build updatable data structures over Spark RDDs, such as PART [41], but these are greatly limited by the immutability of RDDs.

Streaming and dataflow systems like Spark Streaming [27, 73], Naiad [62], and Flink [36] execute queries in real time on streaming data. However, unlike query serving systems, they focus primarily on continuous computation (incrementally updating the result of a query as data comes in) and do not perform data management or low-latency query serving. They are often used to write data into a query serving system.

Cluster management systems like Helix [46], Mesos [48], and YARN [68] are designed to deploy distributed systems at scale. Mesos and YARN are primarily concerned with assigning resources to each application. Helix, like Uniserve,

automatically manages the applications running on it, providing features such as elasticity and fault tolerance. However, it is not designed for query serving workloads and lacks a query model and abstractions for consistency and atomicity.

Auto-sharding systems like Slicer [25], Centrifuge [24], and Shard Manager [55] assign data and queries to shards based on partition keys, like DPA. Slicer and Centrifuge *only* manage key affinity, telling applications what keys are assigned to what servers. Shard Manager goes further and manages data placement, moving data shards between servers. However, unlike Uniserve, these systems do not provide high-level abstractions on top of shards, such as parallel operators and updates with consistency and atomicity guarantees.

Thor [58] stores data in persistent distributed objects for heterogeneous applications to access. These objects resemble DPA actors, but Thor must run object operations on client machines and does not provide high-level abstractions such as a query model or configurable consistency guarantees.

Middleware systems for databases automatically distribute data and queries across existing database installations and provide features like fault tolerance [57, 63] and load balancing [31]. However, these solutions are typically specialized to particular database types, like relational databases [37, 38] or NoSQL stores [44], and do not provide general abstractions to support a wide range of data and query models like DPA.

9 Conclusion

Query serving systems are an important emerging class of distributed systems that power many Internet applications. Traditionally, they are implemented from scratch, requiring substantial effort to add distributed query processing and data management functionality. We presented *data-parallel actors* (DPA), a high-level programming model that allows developers to build reliable and performant distributed query serving systems from single-node data structures and logic. We showed that DPA can express the functionality of a wide range of query serving systems by building a simplified data warehouse and porting Druid, Solr, and MongoDB to DPA in <1K lines of code, matching performance and adding rich missing functionality such as automatic load balancing and auto-scaling. We believe DPA is a valuable tool to help organizations more easily develop these important systems.

Acknowledgments We thank the anonymous reviewers and our shepherd Mahesh Balakrishnan. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, and VMware—as well as Toyota Research Institute, Cisco, SAP, and the NSF under CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. Toyota Research Institute (TRI) provided funds to assist the authors with their research but this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity.

References

- [1] Akka 2.4 migration guide. <https://doc.akka.io/docs/akka/2.4/project/migration-guide-2.3.x-2.4.x.html>.
- [2] Akka transactors documentation. <https://doc.akka.io/docs/akka/2.2/scala/transactors.html>, 2015.
- [3] How to Setup ElasticSearch Cluster with Auto-Scaling on Amazon EC2? <https://stackoverflow.com/questions/18010752/>, 2015.
- [4] MongoDB Cluster with AWS Cloud Formation and Auto-Scaling. <https://stackoverflow.com/questions/30790038/>, 2016.
- [5] Why Architecting for Disaster Recovery is Important for Your Time Series Data. <https://www.influxdata.com/customer/capital-one/>, 2018.
- [6] How Walmart is Combating Fraud and Saving Consumers Millions. <https://www.elastic.co/elasticon/tour/2019/dallas/>, 2019.
- [7] Enterprise Scale Analytics Platform Powered by Druid at Target. <https://imply.io/virtual-druid-summit>, 2020.
- [8] Akka. <https://akka.io/>, 2021.
- [9] Apache Solr. <https://lucene.apache.org/solr/>, 2021.
- [10] Atlas. <https://github.com/Netflix/atlas>, 2021.
- [11] ClickHouse. <https://clickhouse.tech/>, 2021.
- [12] DB-Engines Ranking. <https://db-engines.com/en/ranking>, 2021.
- [13] Elasticsearch. www.elastic.co, 2021.
- [14] InfluxDB. <https://www.influxdata.com/>, 2021.
- [15] MongoDB. <https://www.mongodb.com/>, 2021.
- [16] MongoDB for Analytics. <https://www.mongodb.com/analytics>, 2021.
- [17] OpenTSDB. <http://opentsdb.net/>, 2021.
- [18] Pinecone. <https://www.pinecone.io/>, 2021.
- [19] Shards and Indexing Data in SolrCloud, Aug 2021.
- [20] Solr Distributed Requests. https://solr.apache.org/guide/8_8/distributed-requests.html, 2021.
- [21] Vespa. <https://vespa.ai/>, 2021.
- [22] YCSB GitHub. <https://github.com/brianfrankcooper/YCSB>, 2021.
- [23] Ray collective communication. <https://docs.ray.io/en/latest/ray-collective.html>, 2022.
- [24] Atul Adya, John Dunagan, and Alec Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *NSDI*, volume 10, pages 1–16, 2010.
- [25] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: Auto-sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, 2016.
- [26] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.
- [27] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD ’18*, page 601–613, New York, NY, USA, 2018. Association for Computing Machinery.
- [28] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394, 2015.
- [29] Joe Armstrong. A History of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pages 6–1, 2007.
- [30] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, September 2010.
- [31] Jaiganesh Balasubramanian, Douglas C Schmidt, Lawrence Dowdy, and Ossama Othman. Evaluating the Performance of Middleware Load Balancing Strategies. In *Proceedings. Eighth IEEE International Enterprise Distributed Object Computing Conference, 2004. EDOC 2004.*, pages 135–146. IEEE, 2004.
- [32] Jeff Barr. New AWS Auto Scaling – Unified Scaling For Your Cloud Applications. 2018.

- [33] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.
- [34] Andrzej Bialecki, Robert Muir, and Grant Ingersoll. Apache Lucene 4. In *SIGIR 2012 Workshop on Open Source Information Retrieval*, page 17, 2012.
- [35] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.
- [36] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
- [37] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-Based Database Replication: the Gaps Between Theory and Practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 739–752, 2008.
- [38] Emmanuel Cecchet, Marguerite Julie, and Willy Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference*, number CONF, 2004.
- [39] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
- [40] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, et al. Unicorn: A system for searching the social graph. *Proceedings of the VLDB Endowment*, 6(11):1150–1161, 2013.
- [41] Ankur Dave, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Persistent adaptive radix trees: Efficient fine-grained updates to immutable data.
- [42] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. 2004.
- [43] Tamer Eldeeb and Phil Bernstein. Transactions for distributed actors in the cloud. Technical Report MSR-TR-2016-1001, October 2016.
- [44] Felix Gessert, Florian Bücklers, and Norbert Ritter. Orestes: A scalable database-as-a-service architecture for low latency. In *2014 IEEE 30th international conference on data engineering workshops*, pages 215–222. IEEE, 2014.
- [45] Mainak Ghosh, Ashwini Raina, Le Xu, Xiaoyao Qian, Indranil Gupta, and Himanshu Gupta. Popular is Cheaper: Curtailing Memory Costs in Interactive Analytics Engines. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.
- [46] Kishore Gopalakrishna, Shi Lu, Zhen Zhang, Adam Silberstein, Kapil Surlaker, Ramesh Subramonian, and Bob Schulman. Untangling cluster management with helix. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC ’12, New York, NY, USA, 2012. Association for Computing Machinery.
- [47] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1917–1923, 2015.
- [48] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.
- [49] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.
- [50] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin*, 2012.
- [51] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jiali Li, et al. Pinot: Realtime OLAP for 530 Million Users. In *Proceedings of the 2018 International Conference on Management of Data*, pages 583–594, 2018.
- [52] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.
- [53] Rajesh K Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference*

- on Principles and Practice of Programming in Java*, pages 11–20, 2009.
- [54] Xavier Léauté. Benchmarking Druid. 2014.
- [55] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard manager: A generic shard management framework for geo-distributed applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP ’21, page 553–569, New York, NY, USA, 2021. Association for Computing Machinery.
- [56] Roman Leventov. The Challenges of Running Druid at Large Scale, Nov 2017.
- [57] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware Based Data Replication Providing Snapshot Isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 419–430, 2005.
- [58] Barbara Liskov, Atul Adya, Miguel Castro, Sanjay Ghemawat, R Gruber, U Maheshwari, Andrew C Myers, Mark Day, and Liuba Shrira. Safe and efficient sharing of persistent objects in thor. *ACM SIGMOD Record*, 25(2):318–329, 1996.
- [59] Michael McCandless. Lucene nightly benchmarks. 2020.
- [60] Gian Merlino. Druid Initial Join Support, Oct 2019.
- [61] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.
- [62] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.
- [63] Marta Patiño-Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems (TOCS)*, 23(4):375–423, 2005.
- [64] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.
- [65] Mark Raasveldt. MonetDBLite: An Embedded Analytical Database. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1837–1838, 2018.
- [66] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. Ieee, 2010.
- [67] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.
- [68] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O’Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC ’13, New York, NY, USA, 2013. Association for Computing Machinery.
- [69] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for fine-grained tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686. USENIX Association, April 2021.
- [70] Fangjin Yang, Eric Tscherter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A Real-Time Analytical Data Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 157–168, 2014.
- [71] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2009.
- [72] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the*

- 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.
- [73] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, page 423–438, New York, NY, USA, 2013. Association for Computing Machinery.
- [74] Siyuan Zhou and Shuai Mu. Fault-tolerant replication with pull-based consensus in mongodb. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 687–703. USENIX Association, April 2021.

Orca: Server-assisted Multicast for Datacenter Networks

Khaled Diab Parham Yassini Mohamed Hefeeda

*School of Computing Science
Simon Fraser University
Burnaby, BC, Canada*

Abstract

Group communications appear in various large-scale datacenter applications. These applications, however, do not currently benefit from multicast, despite its potential substantial savings in network and processing resources. This is because current multicast systems do not scale and they impose considerable state and communication overheads. We propose a new architecture, called Orca, that addresses the challenges of multicast in datacenter networks. Orca divides the state and tasks of the data plane among switches and servers, and it partially offloads the management of multicast sessions to servers. Orca significantly reduces the state at switches, minimizes the bandwidth overhead, incurs small and constant processing overhead, and does not limit the size of multicast sessions. We implemented Orca in a testbed to demonstrate its performance in terms of throughput, consumption of server resources, packet latency, and the impact of server failures. We also implemented a sample multicast application in our testbed, and showed that Orca can substantially reduce its communication time, through optimizing the data transfer between nodes using multicast instead of unicast. In addition, we simulated a datacenter consisting of 27,648 hosts and handling 1M multicast sessions, and we compared Orca versus the state-of-art system in the literature. Our results show that Orca reduces the switch state by up to two orders of magnitude, the communication overhead by up to 19X, and the control overhead by up to 14X, compared to the state-of-art.

1 Introduction

Many modern datacenter applications require group communications in the form of one-to-many or many-to-many patterns. Examples of these applications include distributed databases, telemetry systems, consensus protocols, and machine learning systems. Multicast can efficiently support these communication patterns. For example, in distributed databases, multicast can be used to distribute and replicate data among servers [69]. For telemetry systems, multicast is

suitable for sending updates and monitoring data to collector nodes [46, 67]. In addition, multicast can be used for state machine replication tasks in the Paxos consensus protocol and its variations [21, 39, 45, 53]. Furthermore, multicast can improve the performance of iterative algorithms that distribute data from a server to multiple working nodes. Examples of such algorithms appear in training machine learning models [28], text mining [48], and recommendation systems [36].

In addition to the above applications, an efficient multicast primitive would benefit various systems that naturally perform group communication. For example, publish-subscribe systems [37, 58, 68] typically send each message to a group of receivers. These systems are the substrate for many applications such as activity trackers, log aggregators, and stream processing frameworks. Moreover, in the emerging serverless platforms [2, 60], a common pattern is that a worker communicates with multiple other workers to enroll them in a single burst computation [7, 8], which can efficiently be realized using multicast.

Despite its potential significant bandwidth savings, multicast faces multiple challenges that slow down its deployment by major cloud providers [62]. First, to forward packets on links belonging to the multicast tree, multicast forwarding requires *maintaining state at all switches* for each session, which imposes substantial memory overheads on switches. Second, *updating and refreshing this state* upon changes generates a storm of messages, which reduces the scalability of switches as they are required to process numerous control packets. Finally, since multicast trees in datacenters could potentially span many switches and servers, *encoding these trees into labels* could impose substantial processing, communication and/or bandwidth overheads.

As a result, there has been a lack of efficient and scalable multicast systems that support large numbers of sessions. For example, in practice, switch vendors are forced to limit the number of IP multicast sessions per switch [55], because of the inefficiencies introduced by the group management [50] and tree construction [16] protocols of IP multicast. This is not cost-effective for cloud providers. In addition, while

current datacenter multicast approaches, e.g., [5, 6, 32, 43], improve upon the basic IP multicast, they also do not scale well and impose substantial overheads on the network, as we show in this paper. To partially mitigate the lack of efficient multicast systems, many datacenter applications had to rely on (inefficient) application-layer protocols [51]. For example, Apache Spark [40] implements its own primitives [9] such as Cornet [36] and HTTP-based multicast.

This paper presents a new architecture, called Orca, to realize efficient multicast forwarding that can support millions of concurrent multicast sessions in datacenter networks. The idea of Orca is to *offload* some of the state maintained at network switches to end servers. To achieve this idea, Orca computes *fixed-size and compact* labels and attaches them to packets of multicast sessions. These labels effectively enable shifting some of the data plane tasks to servers. As a result, Orca significantly reduces the state at switches, minimizes the bandwidth overhead, incurs small and constant processing overhead, does not limit the size of multicast sessions, and eliminates redundant traffic. Realizing the proposed *server-assisted* multicast approach, however, faces multiple challenges at the control and data planes that Orca addresses. At the control plane, the proposed architecture needs to calculate optimized labels, manage state at servers, and handle failures. At the data plane, it requires packet processing algorithms at switches and servers that sustain the line-rate performance and minimize the latency and resource consumption.

This paper makes the following contributions.

- We introduce the idea of *server-assisted (or offloaded) multicast* for scalable multicast services in datacenters.
- We design a hierarchical control plane that efficiently manages multicast sessions and their dynamics, handles network failures, and does not impose high control overheads (§3.3 and §3.4).
- We present a scalable data plane algorithm to process multicast packets within high-speed datacenter networks, without introducing redundant traffic or requiring switches to maintain large states (§3.5).
- We design and implement APIs to transparently integrate multicast into datacenter applications (§4).
- We implement the proposed multicast approach and evaluate its performance in a testbed using programmable switches to demonstrate its practicality (§5). Our results show that the proposed approach can support high-speed traffic, uses small CPU resources at servers, and imposes small and predictable packet delays.
- We show the potential significant gains achieved by using multicast instead of unicast in datacenter applications. We implemented a sample application using Orca and the unicast approach used in current systems such as Apache Spark [40]. For this application that has only 12 receivers, our results show that Orca can reduce the

communication time by almost an order of magnitude; larger savings are expected for applications with more receivers. In addition, since an Orca sender transmits only one copy per packet regardless of the number of receivers in the session, the required CPU resources are significantly reduced, compared to using unicast.

- We compare Orca against the closest system in the literature, Elmo [5], in large-scale simulations (§6). Our results show that Orca reduces the switch state by up to two orders of magnitude, the communication overhead by up to 19X, and the control overhead by up to 14X compared to Elmo in large-scale datacenter networks.

2 Related Work

Internet Multicast. IP multicast is not practical for datacenter networks because of its limited scalability for both the control and data planes [11, 52]. Specifically, its group management and tree construction protocols, e.g., IGMP [50] and PIM [16], need to maintain state at routers belonging to each multicast session. Moreover, to refresh this state, these protocols generate control messages that routers need to process. These overheads limit the number of multicast sessions, and they could delay a receiver joining a session for up to 23 seconds [66], which is not practical for datacenters. Furthermore, current multicast approaches designed for ISP networks, e.g., [1, 43], introduce significant communication overheads, and thus they are not suitable for datacenter networks.

Datacenter Multicast. Multiple approaches, e.g., [5, 32, 35], attempted to address the challenges of IP multicast. Li et al. [35] propose a multi-class Bloom filter (MBF) to support multicast in datacenter networks. For every interface, MBF uses a Bloom filter to store whether packets of a session should be duplicated on that interface. MBF may introduce redundant traffic due to the probabilistic nature of Bloom filters. Li and Freedman [32] partition the IP address space and aggregate addresses for different sessions. However, this approach consumes the limited flow table resources in switches and limits the number of supported multicast sessions. Elmo [5] encodes links of a multicast tree into rules to be attached to packets and maintained at switches. Elmo is the state-of-art multicast system for datacenters, and we compare against it.

Other works, e.g., [9, 10, 25], enabled multicast in circuit-switched datacenter networks. For example, Republic [9] and Blast [25] realize multicast by using additional optical circuit switches. Orca is designed to be deployed in the common packet-switched networks. Application-layer multicast approaches could also be used in datacenters, by concurrently sending unicast flows to multiple receivers. This, however, results in inefficient bandwidth utilization [47, 49, 51], and increases the CPU load on the sender.

Server-assisted Data Planes. While Orca is the first server-assisted multicast for datacenters, to the best of our knowledge,

it is not the first work to utilize server resources to implement parts of the data plane. For examples, Katta et al. [17] propose an OpenFlow [26] rule caching system using both switch TCAM and server memory, which is managed by a controller. In contrast, we design Orca to reduce the state maintained at switches instead of improving how the large number of rules are stored. A recent work [4] offloads the state of network functions to the server memory using RDMA. Unlike this system, Orca has small header sizes and its agents maintain small state instead of large network function state. Moreover, Orca simplifies how state is fetched, managed, and replicated.

3 Orca: Server-assisted Multicast

We start this section by specifying the design goals of Orca. Then, we present an overview of Orca describing its main components and how they work together. This is followed by the details of each component. In the **Appendix**, we describe various overheads, extensions, and limitations of Orca.

3.1 Design Goals

The objective of this paper is to design a multicast architecture for datacenter networks that achieves the following goals:

- **Reduce State at Switches.** Maintaining large state at network switches not only consumes their scarce memory resources, but it also increases the number and frequency of exchanged update messages to handle network failures and session dynamics. This forces switches to process many control messages while forwarding data packets, which may slow down the data plane [66].
- **Minimize Communication Overhead.** We aim at minimizing the header size per packet to reduce the communication (or bandwidth) overhead, which is critical to decreasing the total transmission time. We note that some of the existing multicast systems, e.g., [5], attach labels that can be as large as the packet payload.
- **Support Large-scale Multicast Sessions.** As datacenter applications become complex, the numbers of multicast sessions and receivers per session are expected to grow at high rates [70]. Existing systems, e.g., [32], do not efficiently scale to support the growing demands and high dynamics of recent datacenter applications.
- **Avoid Redundant Traffic.** Switches should forward packets *only* on links belonging to the multicast tree. This is because redundant traffic wastes network resources and overloads switches. Many of the existing multicast systems, e.g., [5, 35], cannot avoid sending redundant traffic without imposing a substantial amount of communication overheads by using large label sizes and/or increasing the state size maintained at switches.

Simultaneously realizing these goals is challenging as they are inter-dependent and pose conflicting trade-offs. For example, although attaching a large label to packets reduces switch state, it significantly increases the communication overhead and packet processing at switches. Our approach to concurrently achieve these design goals is to attach a *small* and *fixed-size* label to packets of every multicast session. This substantially minimizes the communication overhead and reduces packet processing on switches. In addition, we carefully calculate and process labels to eliminate redundant traffic. Furthermore, to reduce state at switches, we make servers assist in forwarding the packets. As a result, switches will be able to support large-scale multicast sessions.

3.2 Overview

Orca is designed for multi-rooted Clos topologies that are widely deployed in datacenter networks. We use the leaf-spine topology throughout the paper, but the same principles apply for other tree-based topologies. In the leaf-spine topology, the top layer consists of core switches that connect different leaf-spine planes. Spine switches connect leaf switches to other leaf switches and to core switches. Every leaf switch connects a rack of servers to the datacenter network. Each server runs a hypervisor switch and hosts multiple virtual machines (VMs).

In traditional IP multicast, network switches need to maintain state about each multicast session, which imposes significant overheads on the switches. In contrast, the proposed approach *carefully offloads* most of the work needed to manage multicast sessions to end hosts in the datacenter, which enables efficient and scalable multicast—a long standing problem. In addition, unlike IP multicast, Orca uses labels to direct the forwarding of multicast packets through the network. Each label consists of different components, each of which encodes a specific datacenter layer (i.e., leaf, spine, or core). However, as the size of a multicast tree grows, simple stacking of label components would lead to large, *variable-size*, labels and thus significant communication and processing overheads.

The proposed architecture is based on three key insights that enable us to design *small* and *fixed-size* labels and achieve scalability. First, a large portion of the label overhead comes from encoding the tree downstream links belonging to leaf switches, and that this overhead increases for multicast trees with large numbers of receivers. Second, labels belonging to leaf downstream links are not needed until the packet reaches a leaf switch. Third, servers in datacenters already host hypervisor switches to process various packet types. Based on these insights, we logically divide the data plane at the leaf layer: between each leaf switch and the servers connected to it. Then, we offload handling of the leaf downstream labels to some of the servers. To process the labels, these servers run an *Orca agent*, which can run on SmartNICs or CPU cores by integrating it with an existing hypervisor switch or running it as a standalone process.

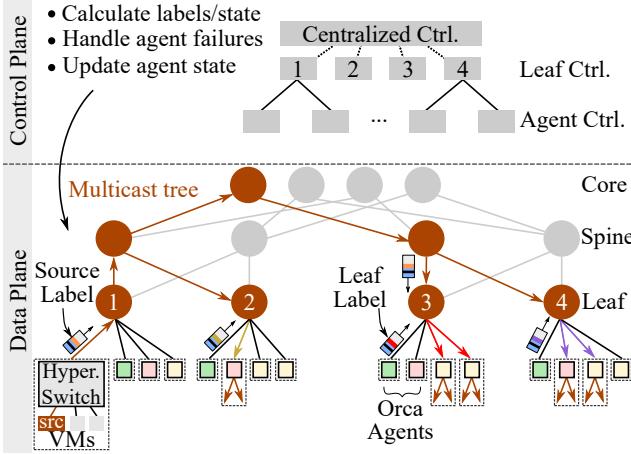


Figure 1: The proposed multicast architecture.

As illustrated in Figure 1, Orca is composed of two components: (i) Hierarchical Control Plane and (ii) Server-assisted Data Plane. The Control Plane is composed of three controllers: Centralized, Leaf, and Agent. The Centralized controller creates labels to represent multicast trees and decides the state that needs to be maintained at network switches; details are presented in §3.3. A leaf controller is deployed on each leaf switch, while agent controllers are deployed on VMs within racks. All three components of the Hierarchical Control Plane collaborate to handle network and agent failures as well as to manage the dynamic nature of multicast sessions, as described in §3.4. The Data Plane, presented in §3.5, instructs switches and Orca agents how to process packets.

At a high-level, a multicast session is created and managed as follows, refer to Figure 1. The tree spanning the source and receivers has one path from the source VM to any core switch, then it reaches the receivers by branching to spine and leaf switches. The tree is then given to the centralized controller, which creates a fixed-size label (referred to as *source label*) to represent a part of the tree. It is important to notice that although the multicast tree can be large and spans many parts of the datacenter network, Orca optimizes the source label and keeps its size small and constant, as described in §3.3. The source label is sent to the source of the session, which attaches it to each packet. The packet is then sent upstream to spine and core switches, which forward it based on various components of the source label in that packet. Then, the packet is sent downstream from the spine and core switches, using other components of the source label, to the leaf switches that have receivers of the session in their racks. Each leaf switch sends the packet to an active Orca agent within its rack. The agent replaces the source label with another label (called *leaf label*) and sends it back to the leaf switch. The leaf label contains the information needed by the leaf switch to forward the packet to the end receivers within the rack.

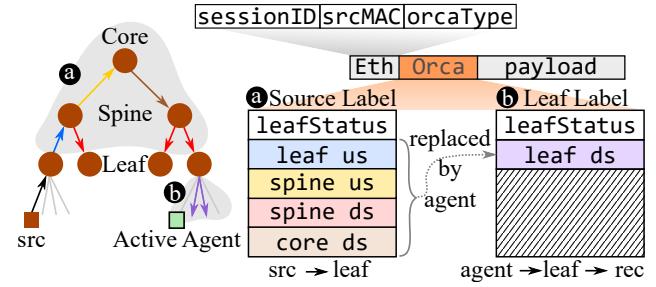


Figure 2: Structure of the Orca header. The color of each label component matches the corresponding link in the network.

3.3 Calculating Labels

Labels play a critical role in the proposed multicast architecture, and they need to be carefully designed to ensure proper functioning of the multicast system as well as minimize the communication and processing overheads.

The centralized controller computes a fixed-size source label that consists of *four components* and a single leafStatus bit. Figure 2 illustrates the header format of an Orca data packet and the label structure. The four label components encode tree links belonging to leaf upstream (us), spine upstream, spine downstream (ds), and core downstream links. When a data packet reaches an active agent, the source label is replaced with the corresponding leaf label to forward packets to the multicast receivers.

The centralized controller calls the CALCULATELABELS algorithm (pseudo code is given in Algorithm 1) to calculate a source label to be attached to packets of the multicast session, a set of leaf labels to be maintained at the agents and state to be maintained at spine switches (if needed). No session state is needed at core or leaf switches. The algorithm takes as input the multicast tree \mathbb{T} . It groups tree links of each network layer and encodes their IDs independently in a fixed-size label component.

The algorithm first creates a bitmap of size (in bits):

$$1 + \max(L_d, \lceil \log(L_u) \rceil + P_d + F + \lceil \log(P_u) \rceil + C_d),$$

where L_d , L_u , P_d , P_u , and C_d are the maximum numbers of downstream and upstream ports per leaf switch, downstream and upstream ports per spine switch, and downstream ports per core switch. F is the size of a filter encoding the spine downstream links. This bitmap accommodates the leaf labels that will be inserted by agents. A typical datacenter switch has 48 ports [5]. Thus, the size of Orca label is 19 bytes in most practical cases.

The first bit in an Orca source label is the leafStatus bit, which indicates whether an agent has replaced a source label with a leaf label. The remaining bits are used to encode links of the multicast tree based on four cases as follows.

Case 1: Leaf and Spine Upstream. For the leaf and spine upstream links, the CALCULATELABELS algorithm maps the two link IDs to outgoing ports in the leaf and spine

Algorithm 1 The CALCULATELABELS algorithm.

Input: \mathbb{T} : multicast tree
Output: L : computed source label sent to the source VM
Output: \mathbb{S} : state sent to a subset of the spine switches
Output: \mathbb{F} : set of leaf labels, each is sent to an agent

```
1: function CALCULATELABELS( $\mathbb{T}$ )
2:    $\langle L, \mathbb{S} \rangle = \text{CALCULATESOURCELABEL}(\mathbb{T})$ 
3:    $\mathbb{F} = \text{CALCULATELEAFLABELS}(\mathbb{T})$ 
4:   return  $\langle L, \mathbb{S}, \mathbb{F} \rangle$ 
5: function CALCULATESOURCELABEL( $\mathbb{T}$ )
6:   size = 1 + max( $L_d, \lceil \log(L_u) \rceil + P_d + F + \lceil \log(P_u) \rceil + C_d$ )
7:    $L = \text{BitString}(size)$ 
8:    $L.append(0)$  // Set leafStatus to 0 (source label)
9:   Case 1: Leaf and Spine Upstream.
10:   $L.append(\mathbb{T}.leaf\_us\_link().port\_num)$ 
11:   $L.append(\mathbb{T}.spine\_us\_link().port\_num)$ 
12:   Case 2: Spine Downstream.
13:   // Common downstream ports across spine switches
14:    $\mathbb{C} = \text{FINDCOMMONPORTS}(\mathbb{T}.spine\_switches())$ 
15:    $L.append(\text{MAPTOBITSTRING}(\mathbb{C}, P_d))$ 
16:   // Call Algorithm 2
17:    $\langle D, \mathbb{S} \rangle = \text{ENCSPINEDSLINKS}(\mathbb{T}, \mathbb{C}, F)$ 
18:    $L.append(D)$ 
19:   Case 3: Core Downstream.
20:    $core\_links = \mathbb{T}.core\_ds\_links()$ 
21:    $L.append(\text{MAPTOBITSTRING}(core\_links, C_d))$ 
22:   return  $\langle L, \mathbb{S} \rangle$ 
23: function CALCULATELEAFLABELS( $\mathbb{T}$ )
24:    $\mathbb{F} = \{\}$ 
25:   Case 4: Leaf Downstream.
26:   for ( $leaf \in \mathbb{T}.leaf\_switches()$ ) do
27:     // Each bit set to 1 represents a session receiver
28:      $lbl = \text{MAPTOBITSTRING}(leaf.ds\_links(), L_d)$ 
29:      $\mathbb{F} = \mathbb{F} \cup lbl$ 
30:   return  $\mathbb{F}$ 
```

switches and encodes these port numbers as two labels of sizes $\lceil \log(L_u) \rceil$ and $\lceil \log(P_u) \rceil$ bits, respectively.

Case 2: Spine Downstream. Since the multicast tree may include more than one spine switch, reserving a bit per spine downstream link significantly increases the label size. Instead, we trade off large label sizes, which impose overhead on every single multicast packet, with a small state maintained at a subset of the spine switches. Specifically, we encode the spine downstream links using two label components with a total size of $P_d + F$ bits. The first label component encodes the common downstream ports across all spine switches belonging to the multicast tree using P_d bits. For example, if a tree has three spine switches and the first two outgoing ports belong to the tree for each of the three spine switches, then the calculated label is 1100...0. We refer to this set of common ports as \mathbb{C} .

The second label component uses a probabilistic set membership data structure (a.k.a *filter*) to encode the remaining spine downstream links in a label D of size F bits. Since these filters trade off membership accuracy for space efficiency, they may result in false positives, which occur when some spine downstream links that do not belong to the multicast tree are incorrectly included in the computed filter. False positives result in redundant traffic. To address this issue, we calculate a state alongside the label. This state can have zero or more entries, and each entry takes the form $\langle sID, linkID \rangle$, where sID is the ID of the spine switch that should maintain this state and $linkID$ is the ID of the downstream link identified as a false positive during the encoding. The filter supports two functions: (i) $D = \text{encode}(l)$ to encode an input item l (link ID in our case) into a bit string D of size F bits using a hash function, and (ii) $\text{check}(l, D)$ to check whether a given item l belongs to D using the same hash function. Our link encoding algorithm can use any filter, e.g., Bloom [54] and Cuckoo [27] filters, that can support: (1) adding an item to an existing filter, (2) testing whether an item exists (potentially with false positives), and (3) avoiding false negatives. A false negative happens when a link in the multicast tree is not represented in the filter.

The CALCULATELABELS algorithm calls the ENCSPINEDSLINKS function, the pseudo code is shown in Algorithm 2 in the Appendix. This function encodes spine downstream links of the multicast tree into a label D and calculates the state \mathbb{S} to be maintained by spine switches. To calculate \mathbb{S} , we need to identify false positive links belonging to spine switches. We refer to the subset of the spine downstream links that *may* be false positives as *candidates*. There are two conditions for a spine downstream link to be a false positive candidate. First, it has to be attached to a spine switch that belongs to the multicast tree, as packets of that session do not reach other spine switches. Second, it should not belong to the spine downstream links of the multicast tree. Otherwise, it is not a false positive.

The ENCSPINEDSLINKS function has three steps. First, it encodes every link l in the set of spine downstream links using the *encode* function. Then, it computes the false positive candidates based on the two conditions mentioned earlier. Finally, it calculates the state that needs to be maintained at spine switches by checking all false positive candidates stored in *cands* and adding only the links that collide with the spine downstream links encoded in D and not belonging to \mathbb{C} .

Case 3: Core Downstream. The CALCULATELABELS algorithm maps IDs of core downstream tree links to a bitmap of size C_d bits, where C_d is the maximum number of downstream ports in the core layer. The label bits identify the outgoing ports at the core switch belonging to the multicast tree. Thus, a bit at location i in the label is set to 1 if the core switch should duplicate packets on the i^{th} port.

Case 4: Leaf Downstream. For every leaf switch belonging to the multicast tree, the CALCULATELABELS algorithm

calculates a leaf label that encodes all link IDs to reach the receivers of the session within the rack managed by that leaf switch. Each leaf label simply maps the link IDs into a bitmap of size L_d bits.

3.4 Handling Session Dynamics and Failures

Orca employs simple, but effective, mechanisms to manage the dynamic nature of multicast sessions, and to mitigate network and agent failures. We only assume the continuous availability of the top-level, centralized controller of Orca, which can be achieved through mechanisms usually used for such datacenter functions, e.g., [42].

Session Dynamics. Multicast receivers can join and leave any time during the sessions by calling corresponding APIs that communicate with the centralized controller (§4).

When a joining/leaving event is received, the centralized controller runs a simple method (Case 4 in §3.3) to update leaf labels at the agents. The controller then sends the updated leaf labels to the corresponding leaf controllers. The message also includes a unique sequence number. Each leaf controller relays the new leaf label to all active and standby agents within its rack. An agent updates its memory with the new label if the received sequence number is larger than the largest sequence number it has processed so far. Agents then send confirmation messages to upstream controllers indicating that the new changes were processed successfully.

Orca Agent Failures. In each rack, we maintain N Orca agents active and M as standby, where N, M are configurable parameters. All agents within a rack maintain the same leaf label per multicast session. The leaf switch in the same rack distributes the labeling workload among the N active agents, in a round robin manner. This adds more reliability and reduces the labeling load on individual agents.

All agents, active and standby, send heartbeat packets to the leaf controller at a fixed rate. If the leaf controller does not receive any heartbeats from an agent within a timeout period T , the agent is assumed failed. T is in the same order of the RTT within a single rack, which is often a few milliseconds [19]. If the failed agent was active, the leaf controller replaces it by one of the standby agents, otherwise the controller just removes the failed agent from the standby set.

We note that Orca agents deployed in a rack operate independently of agents deployed in other racks. Thus, our approach *localizes* failures within each rack, which reduces the control overhead and increases the control plane responsiveness. In other words, a leaf controller handles *only* the failures of its downstream agents. In addition, heartbeats provide responsiveness and simplicity, which is sufficient in our system as all agents maintain the same state. The state is updated across all agents when there is a change in the multicast tree, which is detected by the centralized controller.

Network Failures. The centralized controller detects network (link and switch) failures using existing systems such as [12].

Once a failure is detected, the controller re-calculates new source and leaf labels for the impacted sessions (§3.3). It also computes a new state at switches (if needed). To mitigate losses during network or agent failures, applications can use reliable transport protocols, e.g., [9].

3.5 Server-assisted Data Plane Forwarding

The data plane in Orca consists of leaf, spine and core switches, as well as agents deployed at servers. The data plane components process received packets as described below.

Leaf Switch. For a packet received on a downstream port, the leaf switch data plane processes that packet based on the `leafStatus` bit. If this bit is zero, i.e., a packet from the source, the data plane reads the first $\log(\lceil L_u \rceil)$ bits after the `leafStatus` bit as a leaf upstream label component, and forwards the packet based on the upstream port number encoded in that component. If the `leafStatus` is set to 1, this means the active agent has inserted a leaf label into the packet header. Thus, the data plane uses the leaf label component of size L_d bits to forward/duplicate the packet to corresponding servers. Specifically, a bit set at location i instructs the data plane to duplicate the packet on its i^{th} port.

If a packet is received on an upstream port, the data plane forwards the packet on a port connected to one of the active agents, which is set and updated by the leaf controller.

Spine Switch. For a packet received on a downstream port, the data plane processes both the upstream and downstream label components. First, the packet is forwarded to a core switch by reading the spine upstream label, which encodes the outgoing port number. Second, since the packet may be forwarded/duplicated on the spine downstream links, the data plane runs the `PROCSPINEDSLABEL` algorithm to process the two spine downstream labels (pseudo code is shown in Algorithm 3 in the Appendix). This algorithm is executed for packets received on upstream ports as well. The algorithm first identifies the common links \mathbb{C} by reading the first label. If a link is set to one in the label, the switch duplicates the packet on that link. Then, the algorithm uses the second label D and state `State` maintained by the spine switch to decide which of the other downstream links belong to the tree.

For each link $l \notin \mathbb{C}$, the algorithm decides to not forward the packet on l if it is not encoded in D . This is because filters in Orca do not produce false negatives. When $l.id$ exists in the label, the algorithm needs to check the maintained state `State` as $l.id$ may be a false positive. Recall that the state contains the false positive links computed by the control plane. The algorithm duplicates the packet only if $l.id$ does not exist in `State[sID]`.

Core Switch. The data plane reads the core downstream label component (of size C_d bits) to forward/duplicate the packet to downstream spine switches. Similar to the leaf downstream label, this label encodes which downstream ports the incoming packet should be forwarded on.

Orca Agent. For incoming packets from a leaf switch, the agent data plane checks the `leafStatus` bit. If it is set to 0 (i.e., it has no leaf label), the agent reads the corresponding leaf label from the leaf label map and inserts it into the packet header and sets the `leafStatus` bit to 1. If the `leafStatus` bit is 1, this packet is destined to receiver VMs.

Overheads and Limitations of Orca. We describe Orca overheads and limitations in the Appendix. In summary, Orca agents require small processing resources at servers as the computation performed on packets is simple. In addition, Orca adds a small latency to packets at the leaf layer only. Furthermore, deploying Orca in graph-based datacenter networks requires changes in some of its components.

4 Implementation and Orca APIs

We briefly describe the implementation of Orca components which are illustrated in Figure 3.

Orca APIs for Multicast Applications. We implemented two sets of interfaces for multicast applications. The first one is between the agent and applications to provide `send` and `recv` functionalities seamlessly to the application. These APIs use Unix domain sockets to communicate with the agent. When using `send` at the source, the agent gets data from the sockets, attaches Orca label and transmits the packets to the leaf switch. Receivers use `recv` to instruct the agent to relay available data to the application. The second set of APIs is between applications and the centralized controller to create, join and leave multicast sessions. We implemented the communication and data encoding/decoding using gRPC [64] and Protocol Buffers.

Orca Agents. We implemented the agent using BESS [23,61] in about 640 lines of C++, where packet processing is done completely in the user space using DPDK [63]. The agent leverages Receive Side Scaling (RSS) to receive packets on different RX queues, each is assigned to a single core.

Orca Hierarchical Controller. The centralized and leaf controllers are implemented in about 3K lines of Golang. The current implementation of the leaf controller communicates with the data plane through raw or Unix domain sockets, but it can easily support other interfaces. For instance, in our testbed, the leaf controller process is deployed to the workstation hosting a NetFPGA, and it uses PCIe to exchange control packets with the NetFPGA, which is done through the RIFFA framework [18]. Communications between the centralized and leaf controllers are done using gRPC [64].

Switch Data Plane. Since Orca’s data plane processing is simple, it can easily be implemented in different programmable switches. We implemented the data plane of Orca in NetFPGA SUME [29] and tested it on multiple of them. We used the open source project in [56], and implemented a Verilog module to decide the outgoing ports. We measured the number of clock cycles and resource usage of our implementation using Xilinx tools. Our implementation of core and leaf

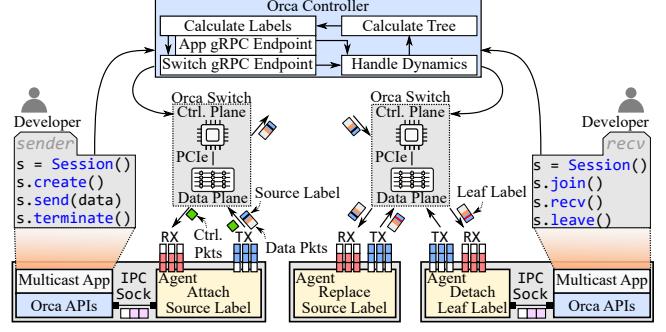


Figure 3: Implementation of Orca components.

switches maps the corresponding bitmaps to outgoing ports, which is done in one clock cycle. We implemented the spine switch algorithm in three clock cycles to identify common ports and check the Bloom filter using a bitwise-AND between the label and hashed link IDs stored at the switch, read state from memory, and decide the outgoing ports. In terms of resource usage, our algorithm utilizes a tiny percentage of the available hardware resources. It uses 0.12% and 0.16% of the available lookup tables (LUTs) and registers, respectively.

5 Evaluation of Orca in Testbed

We evaluate Orca in a testbed to demonstrate its potential benefits to applications and assess the performance of its data plane and control plane components. The testbed has three NetFPGA SUME switches [29] representing a spine and two leaf switches, each of which has four 10GbE ports. The testbed also has five workstations to act as Orca agents and multicast senders and receivers. We configure our testbed to only have one active agent per rack to stress our labeling algorithm at servers. Each workstation is equipped with a dual-port Intel 82599ES 10GbE NIC. Each leaf switch is connected to two workstations, and the spine switch is connected to one workstation. We generate traffic at line rate from a multicast source and transmit it to leaf switches through the spine switch.

5.1 Benefits of Orca

We implemented a sample multicast application that has the same behavior of the iterative machine learning algorithms implemented in Spark [40]. In these algorithms, the data to be processed is often written to files, and a server iteratively sends them to all receivers for processing. In our application, a server reads a file and transmits it in chunks. In every iteration, after a file is sent, the server awaits acknowledgment of file reception from the receivers. The next round starts only after receiving all acknowledgments. This emulates the aggregation phase in distributed data processing frameworks [44], which indicates all workers have updated their model parameters. In our implementation, we set the payload size to the maximum

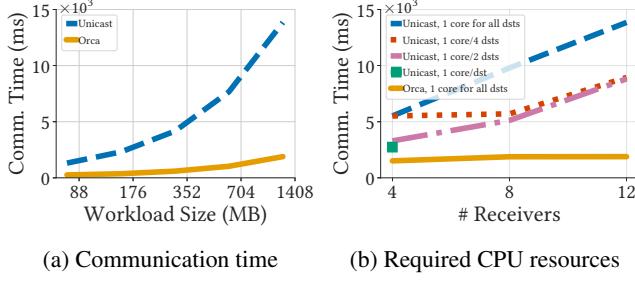


Figure 4: Benefits of Orca.

UDP message length. We run each instance of the client application inside a separate Docker container on the receiver machines.

Performance Metrics. We compare Orca versus the current unicast approach used in systems such as Apache Spark [40]. In particular, we demonstrate the potential benefits of Orca in terms of the communication time, impact of available processing capacity at the sender on the communication time, and total transmitted traffic.

The running time of datacenter applications consists of communication and computation times. The communication time of an application is the total time spent on sending data and receiving corresponding acknowledgments without including the computation times. We measure the communication time of Orca versus unicast, as this is the main aspect being optimized by Orca and it does not control or modify the computations. In addition, we aim at showing that Orca can present the same packet to the application layer much faster compared to the current unicast approaches.

Depending on the application and its total computation time, Orca can reduce the running times of a variety of applications. For example, the authors in [36] reported that the communication time of data-intensive tasks using unicast can be larger than the computation time, especially as the number of workers increases. Thus, optimizing data transfer is critical for these applications.

Workloads. The number and size of the transmitted files are similar to the ones used in the distributed latent Dirichlet allocation (LDA) algorithm [9, 30]. LDA identifies topics in the input documents and maps each document to a set of topics. The vocabulary training set is the data transmitted to the worker nodes. To calculate the workload size, we run the algorithm on a synthetic dataset containing 16,923 documents and 100 topics using the tool in [71]. To evaluate Orca using realistic workloads, we create five different workloads with sizes of 88MB, 176MB, 352MB, 704MB, and 1.4GB.

Results. We conduct experiments using concurrent 4, 8 and 12 receivers and the five different workloads mentioned above.

Figure 4a shows the communication time for Orca and unicast for different workloads when the number of receivers is 12. The sender in the multicast session uses one CPU core to transmit the traffic. These results show that Orca can

significantly reduce the communication time for all considered workloads. In addition, the figure shows that, unlike the case for Orca, the communication time for unicast grows in a super-linear manner with the workload size. This is because the unicast sender needs more time to transmit packets to each of the concurrent 12 receivers, whereas Orca transmits only a single packet for all receivers. Packet transmission at high rates also requires processing cycles.

To analyze the impact of the available processing capacity at the sender on the communication time, we allocate a varying number of CPU cores to transmit the traffic of the multicast session in the case of unicast. For Orca, only one CPU core is used. In Figure 4b, we plot the communication time for the largest workload (1.4GB) for Orca and unicast, as we vary the number of available CPU cores. The figure shows that Orca has a fairly stable communication time as the number of receivers increases, despite using only one CPU core to transmit all packets of the session. In contrast, unicast needs more CPU cores to send the traffic to different receivers to reduce the communication time. In our testbed, allocating a single core per receiver for unicast could not sustain the high packet rate at the sender for 8 and 12 receivers.

Next, we measure the total transmitted traffic from the sender for the largest workload as well as the label overhead of Orca. When using Orca, the total outgoing traffic is only 1.51 GB, compared to 18.01 GB when using unicast. This means the sender in the unicast model would need to transmit 12X more traffic, which not only consumes more bandwidth, but also requires more processing and memory resources to transmit much more packets. The total label overhead of Orca is 7.69 MB which represents only 0.51% of the transmitted multicast traffic.

Although current multicast approaches may yield similar benefits to applications, they cannot scale well to support a large number of multicast sessions. Therefore, we compare the scalability of Orca versus the state-of-art multicast system using large-scale simulations in §6.

Summary: For a sample application with 4–12 receivers, Orca achieves substantial savings in communication time, required processing resources at the sender, and bandwidth, compared to the current unicast approach.

5.2 Data Plane Performance

Throughput of Spine Switches. We report the throughput of the spine switch; we omit the results of leaf and core switches as they run simple forwarding algorithms.

We transmit labelled packets of many concurrent multicast sessions at the maximum link speed (i.e., 10 Gbps) from the source to the spine switch. The labels instruct the spine switch to duplicate packets to two leaf switches. We run this experiment five times for every packet size and compute the average across them. We compare the incoming packet rate against the outgoing packet rates observed at the two leaf

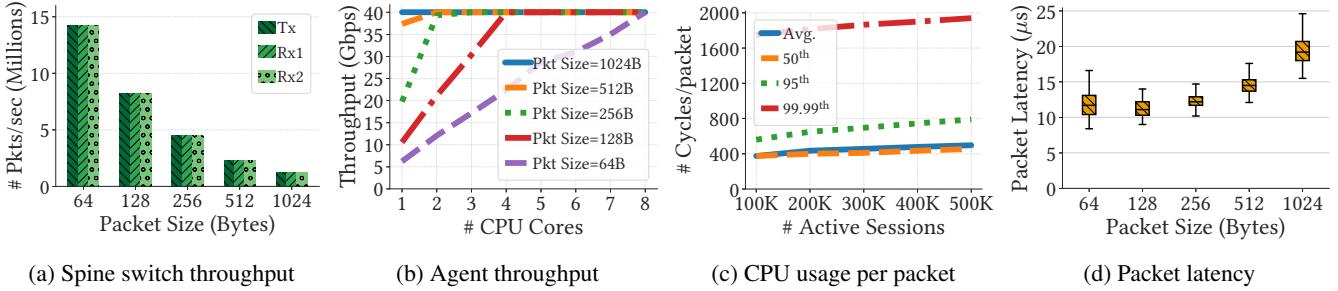


Figure 5: Data plane performance of Orca.

switches in Figure 5a. Our results show that the packet rates are the same (i.e., no packet losses). We also measure the achieved throughput at the two interfaces, and confirm that the spine switch can sustain the 10 Gbps for all packet sizes.

Agent Scalability. We stress and evaluate the scalability of the agent data plane. In this setup, we deploy two NICs (i.e., 4x10GbE ports) at the agent workstation and direct labeled traffic at rate of 40Gbps from the other two workstations. The labels have the `leafStatus` set to zero, which indicates that the agent needs to label them using a corresponding leaf label. We measure the throughput of the packets after being processed by the agent.

Figure 5b shows the throughput versus the number of allocated cores for the data plane, which shows that the agent scales well to support high rates. We measure the smallest packet size at which the agent can sustain the 40Gbps traffic using a single core. Our results show that the agent can sustain this rate using 1 core for packets of size 560 bytes or larger. For enterprise datacenters, the average packet size is reported to be 850 bytes [41]. Furthermore, data-intensive jobs like Hadoop workloads often use 1500-byte packets [20]. That is, Orca agents require only a few cores per rack to support many multicast sessions at high rates. Major datacenters deploy 24–48 servers per rack [14, 65], and each typically has more than 16 cores. That is, even for applications that require small 64-byte packets and send at an aggregate rate of 40Gbps, an Orca agent would need up to 1–2% of the available CPU resources in a rack when SmartNICs are unavailable.

In addition, recall from §5.1 that Orca requires only one CPU core at the sender side regardless of the number of receivers, whereas the current unicast approach needs a proportional number of CPU cores to sustain the transmission rate especially as the number of receivers increases. Thus, the processing capacity needed to run Orca agents will likely be offset by the savings in the processing capacity needed to transmit the traffic in the unicast approach.

Agent CPU Usage. Recall that an active agent needs to look up a leaf label from its memory using the session ID. We measure the total number of CPU cycles needed by the agent to process packets (including labeling and memory lookup). We stress the agent by allocating leaf labels for 1M sessions at

the agent. In this experiment, the sender randomly transmits traffic belonging to a subset of the total sessions, which we refer to as active sessions. We use large numbers of active sessions to stress the agent.

Figure 5c shows different statistics of the used number of CPU cycles per packet (measured by `rdtsc`) when the packet size is 1024 bytes. The results show the efficiency of the agent even without any code optimizations. For example, the agent running on a 3.8GHz CPU needs an average of 99 ns per packet when the number of active sessions is 100K per rack. We note that the number of CPU cycles is constant for different packet sizes, since the agent processes fixed-size labels. To put these numbers in context, existing, optimized, software switches such as OVS [24] and PISCES [15] use 409 and 426 cycles/packet, on average, to handle IP packets, respectively. The average for Orca is 375 cycles/packet when handling 100K active sessions.

Packet Latency and Jitter. We measure the packet latency and jitter of Orca at the leaf layer, which is defined as the total duration from when a packet is sent to the leaf switch to the time it is received by a multicast receiver connected to that switch. We emulate a dynamic traffic scenario, where the sender starts transmitting traffic at 1Gbps and increases the sending rate with 1Gbps steps every 20 seconds until it saturates the link, and holds this transmission rate for another 20 seconds.

Figure 5d shows the packet latency for different packet sizes. For 64-byte packets, the median latency is 11.3 μ s. The packet latency slightly increases for large packet sizes because of the increased transmission time. Notice that in latency-sensitive applications, where latency for individual packets are important, smaller packets are more prevalent [57] where Orca has short packet latency.

We next measure the packet inter-arrival jitter, which is calculated as the difference between the current packet delay and previous packet delay. Our results show that Orca imposes negligible variance in packet latency. The average and maximum inter-arrival jitter values for 1024-byte packets are 1.135 μ s and 3.3 μ s, respectively.

Summary: The Orca data plane is scalable and can sustain high throughputs even with small packet sizes. Orca agents

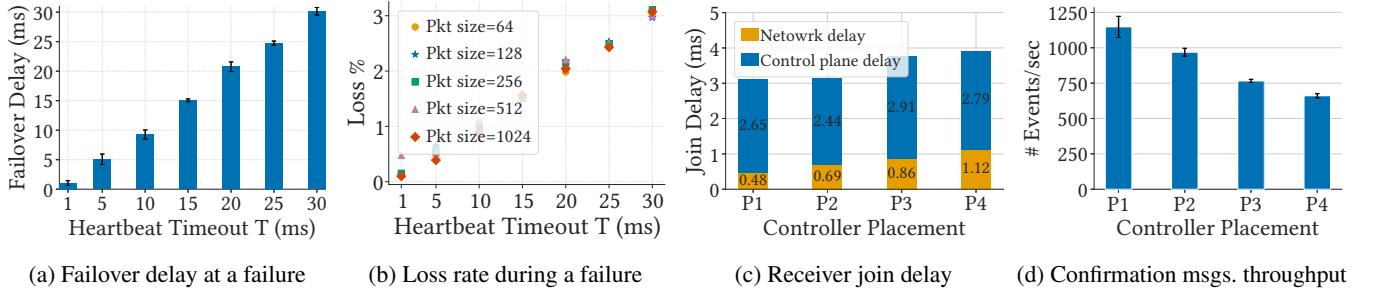


Figure 6: Control plane performance of Orca.

can process large numbers of concurrent sessions, use low CPU resources, and only add a small latency to packets.

5.3 Control Plane Performance

Responsiveness to Agent Failures. Orca localizes agent failures within the rack, thus we analyze failures for a single rack. We measure the performance while an active agent is handling traffic at 10Gbps. We manually crash the active agent and measure the following metrics at a receiver: failover delay, throughput, and data loss rate. We report the results for the worst-case scenario, where the rack has only one active agent. Failover delay is the time when the receiver does not receive traffic due to active agent failure and when it receives traffic again. Figure 6a shows the average failover delay for all packet sizes when we control the heartbeat timeout. The results confirm the fast response of the control plane in choosing a new active agent when the original agent fails. For instance, receivers can resume receiving traffic within 1.04 ms after an active agent fails when T is 1 ms.

We next measure the observed throughput at the receiver during a failure, where we crash the agent after two seconds. For 1024-byte packets and heartbeat timeout of 1ms, we observe a throughput drop by up to 0.012% only. In addition, for all packet sizes, the total throughput drop is less than 0.013% during a failure. Our results confirm that Orca quickly restores the transmission to full capacity after a failure. Finally, we plot the loss rate caused by an agent failure in Figure 6b. When the heartbeat timeout is 1 ms, Orca incurs a loss rate of 0.18%, on average, across all packet sizes. We note that such losses can be easily mitigated by using reliable multicast [9].

Receiver Joining Delay. We assess the performance of the proposed method for updating leaf labels when a new receiver joins. Recall that when a session changes, Orca sends new leaf labels to the corresponding agents. This impacts how quickly a joining receiver would receive traffic. In addition, network delays between the control plane components in datacenters might vary depending on the placement of the controllers and receivers. We emulate different controller placement setups in our testbed by adding synthetic delays at the network interface queues of the workstations (using `tc`) to stress our system.

We consider four different placement setups (P1–P4) starting with no synthetic delay in P1 with mean RTT of 479 μ s and adding 200 μ s of delay every step till we reach a mean RTT of 1,120 μ s (maximum 1,326 μ s) in P4 setup. These RTT values follow what is reported in [19] where the 99th percentile RTT between two hosts is 1.34 ms. We note that even the lowest RTT in our setup (i.e., P1) is larger than the median RTT inside a rack in datacenter networks which is 268 μ s [19].

We measure the receiver join delay, which is the time duration from when a receiver sends a join request for a session and the time the first packet of that session is received by the receiver. For each placement setup, the experiment is repeated for 30 join events. Figure 6c shows the average join delay as well as the contribution of network delays. We report that even in the worst-case scenario (P4), the average join delay is less than 4 ms. In total, the median and 99th-percentile delays are 3.12 ms and 6.53 ms, respectively. To put these numbers into perspective, note that inserting a new rule into an OpenFlow switch takes 1–3 ms, and rule modification delays vary from 2–18 ms [22].

We next measure the throughput of processed confirmation messages at the centralized controller in Figure 6d, which represents the end-to-end performance. In the P1 setup, Orca handles an average of 1,147 msgs/sec (SD is 74). As increasing latency affects gRPC, the average throughput of the largest delay scenario is 662 msgs/sec (SD is 14).

Summary: Orca recovers quickly from failures and it supports dynamic multicast sessions. Furthermore, the failure detection mechanism in Orca is localized to individual racks.

6 Orca versus State-of-Art

We analyze the performance and scalability of Orca and compare it against the state-of-art system, Elmo [5], using large-scale simulations. We use the open-source code of Elmo.

Elmo employs three stages to encode a multicast tree. Elmo encodes switches of a tree as a union of multiple bitmaps representing outgoing ports. When the label size reaches a pre-configured value, Elmo installs forwarding state entries at switches without exceeding their capacities. Otherwise, Elmo calculates a default entry that may result in redundant traffic.

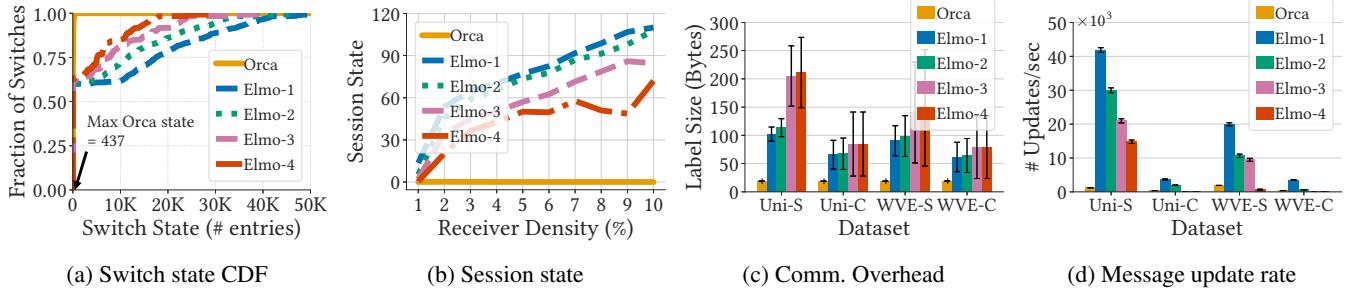


Figure 7: Performance of Orca versus Elmo.

6.1 Simulation Setup

Topology and VM Placement. We simulate a multi-rooted Clos topology consisting of 48 pods, each has 576 48-port leaf and spine switches. This results in a large datacenter network of 27,648 hosts. We use a setup similar to [5, 32]: There are 3,000 tenants, each has a number of VMs ranging from 10 to 5,000. The maximum number of VMs per server is 20. We use two VM placement strategies. The first one is a Clustered placement (denoted by C), which places at most 12 tenant VMs per rack. The second is a Scattered strategy (denoted by S), and it places at most one tenant VM per rack.

Multicast Sessions and Datasets. Multicast receivers per session are randomly chosen from all tenant VMs. The size of these sessions follows two different distributions similar to [5, 32]. The first distribution follows a workload from the IBM WebSphere Virtual Enterprise (WVE) [32], and the second one is a uniform distribution (Uni). The minimum and maximum session sizes for both distributions are 5 and 5,000 receivers, respectively.

We generate four datasets representing various workload characteristics and VM placement strategies. We denote a dataset using its session size distribution and VM placement strategy, e.g., a dataset with uniform session sizes and scattered strategy is referred to as Uni-S. We simulate 1M multicast sessions per dataset, where each tenant has sessions proportional to the number of its VMs.

Orca and Elmo Parameters. We set the filter size in Orca to 69 bits to compute byte-aligned label. For Elmo, we control two parameters to analyze different aspects of it, and set them according to [5]. The first one is the number of rules encoded in Elmo label, which is set to be either 10 or 30. The second parameter is the redundancy limit that controls the amount of redundant traffic caused by sharing a single rule in Elmo label. We set this parameter to be 0 (no redundant traffic) or 12. We refer to the Elmo four configurations as Elmo-1 (10, 0), Elmo-2 (10, 12), Elmo-3 (30, 0) and Elmo-4 (30, 12).

6.2 Data Plane Performance

Switch State. Figure 7a shows the CDF of the switch state for the Uni-S dataset. The results for other datasets are similar.

The figure shows that Orca significantly reduces the state size compared to all considered configurations of Elmo. For example, in Orca, 99% of switches need to only maintain up to 253 entries in their memory, and no switch maintains more than 437 entries. In contrast, for Elmo-1, which calculates the smallest label sizes (i.e., 100 bytes on average), 99% of switches need to maintain up to 47.7K entries in their memory, with some switches need to maintain as many as 53.5K entries. Elmo could not reduce the state even when it doubles the label size. For example, in Elmo-4, 99% of switches need to maintain up to 24K entries in their memory (maximum is 30K entries). This is a significant improvement because it indicates that Orca requires much lower switch memory to support the same number of multicast sessions and much fewer control messages to update the switch state.

We next study the impact of session size on the required state to be maintained for that session in Figure 7b. The figure shows that Orca scales well, and it can reduce the session state by up to 55X compared to Elmo. For example, when a session has 2.5K receivers, Orca needs to maintain state at up to two switches only. Elmo-1, however, needs to maintain state at up to 110 switches.

These significant gains are achieved because, unlike Elmo, Orca does not require maintaining state at *any* leaf or core switch. In addition, the proposed spine labels can encode most of the spine downstream links while requiring small state at few spine switches.

Summary: *Orca reduces state size by up to two orders of magnitude compared to Elmo, and can support a large number of concurrent multicast sessions.*

Communication Overhead. Figure 7c shows that Orca reduces the communication overhead by using a small and fixed-size label of size 19 bytes to forward traffic of 1M sessions. On the other hand, Elmo uses much larger labels. For example, in the Uni-S dataset, the average and maximum label sizes of Elmo-4 are 211 bytes and 368 bytes, respectively; SD is 62 bytes.

Elmo introduces variations in the label size for the *same* configuration across different datasets. This means that changes in VM placement strategy or shifts in traffic patterns introduce unpredictable forwarding performance in Elmo, as

its switches need to process labels with *varying* sizes. For example, changing the VM placement strategy from clustered to scattered in Elmo-4 increases the average label size by 148% because receivers in the scattered strategy span more racks compared to the clustered one. For the same configuration, a shift in traffic pattern from WVE to Uni increases the average label size by 42% as Elmo needs to encode more receivers using the same label size. This is because the WVE distribution is skewed, and thus, fewer sessions have large group sizes compared to the Uni distribution. In contrast, Orca has a *fixed-size* label of 19 bytes because it utilizes the key insights described in §3.2.

Summary: *Orca reduces the communication overhead by up to 19X compared to Elmo while being robust against VM placement strategies and session sizes.*

Redundant Traffic. We define the redundant traffic per session as the ratio between the number of receivers that receive unwanted traffic to the total number of receivers in the session. *By design, Orca does not introduce any redundant traffic.* Elmo may introduce redundant traffic to reduce state size by controlling the redundancy limit parameter as it shares the same rule among multiple switches. We analyze the traffic redundancy of Elmo-2 and Elmo-4 for the Uni-S dataset. Other Elmo configurations have redundancy limit of 0 similar to Orca but they require maintaining much larger state at switches. Our results show that, for Elmo-2, 25% of the sessions have more than 67% redundant traffic, with a maximum value of 172%. For Elmo-4, with much larger labels, the maximum redundant traffic is 113%. That is, the traffic could erroneously be sent to more destinations not participating in the multicast session than the actual receivers.

Summary: *Orca does not introduce any redundant traffic, whereas Elmo may impose up to 172% redundant traffic.*

6.3 Control Plane Performance

Session Dynamics. We randomly generate 1,000 receiver joining/leaving events per second with joining probability of 0.5. Every event changes a multicast tree, and thus, the state maintained at switches may need to be refreshed. Refreshing the state requires the control plane to send update messages to the switches. We measure the total number of update messages per second sent by the controller for both Orca and Elmo. We report the results for all datasets in Figure 7d. For example, the Orca controller needs to send an average of 1,889 messages per second (SD is 45) for the WVE-S dataset. On the other hand, the Elmo controller needs to send 19.9K, 10K, 9.5K and 615 messages per second on average for Elmo-1, Elmo-2, Elmo-3 and Elmo-4, respectively. This is because Orca maintains state only at a small number of switches. Elmo-4 does not need to update many switches. It, however, imposes the largest label size among all Elmo configurations with high amount of traffic redundancy.

Summary: *Orca reduces the rate of update messages by*

up to 10X compared to Elmo.

Network Failures. Similar to session changes, a core or spine switch failure triggers Orca and Elmo to update state at switches if needed. For the WVE-S dataset, Orca needs to send 3,900 messages per core switch failure on average, while Elmo-1, Elmo-2, Elmo-3, and Elmo-4 needs to send an average of 56.2K, 31.2K, 27.6K, and 1.7K messages per core switch failure, respectively. For a spine switch failure, Elmo-1, Elmo-2, Elmo-3, and Elmo-4 send 34.7K, 20.6K, 18K, and 1.2K messages per failure, respectively, whereas Orca sends 4,890 messages per failure. Although Elmo-4 requires sending fewer messages per failure, it imposes significant overheads in terms of the label size and traffic redundancy.

Summary: *Compared to Elmo, Orca reduces the control overhead for handling failures by up to 14X.*

Running Time. Orca calculates labels faster than Elmo. We report the running time of Orca and Elmo in the Appendix.

7 Conclusions and Future Work

We presented Orca, an efficient multicast architecture for datacenter networks. Orca splits the data plane operations between leaf switches and servers. That is, Orca offloads managing multicast sessions from leaf switches to servers. Orca has a scalable control plane that handles session dynamics and network failures. It also has a simple data plane that can sustain high rates and can easily be implemented in programmable switches. The server component in Orca can be implemented on SmartNICs, or on regular CPU cores if SmartNICs are not available. We implemented lightweight APIs to seamlessly integrate multicast into datacenter applications. We also implemented Orca in a testbed that contains programmable switches. We evaluated a sample multicast application in our testbed. Our results show that Orca can substantially reduce the communication time compared to unicast. In addition, we assessed the performance of Orca in terms of its throughput, resource usage, packet latency and the impact of failures. Moreover, we compared Orca versus the state-of-art multicast system, Elmo, using large-scale simulations. Compared to Elmo, Orca reduces the switch state by up to two orders of magnitude and the label size by up to 19X.

This work can be extended in multiple directions. For example, we plan to extend Orca to support various group communication primitives needed by modern datacenter applications.

Acknowledgments

We thank our shepherd, Sujata Banerjee, and the anonymous reviewers for their insightful and helpful comments. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

- [1] Khaled Diab and Mohamed Hefeeda. Yeti: Stateless and generalized multicast forwarding. In *Proc. of USENIX NSDI’22*, Renton, WA, April 2022.
- [2] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proc. of ACM SoCC’21*, pages 138–152, Seattle, WA, November 2021.
- [3] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *Proc. of ACM SIGCOMM’20*, pages 681–693, Virtual Event, August 2020.
- [4] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proc. of ACM SIGCOMM’20*, pages 90–106, Virtual Event, August 2020.
- [5] Muhammad Shahbaz, Lalith Suresh, Jen Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source-routed multicast for public clouds. In *Proc. of ACM SIGCOMM’19*, pages 458–471, Beijing, China, August 2019.
- [6] Toerless Eckert, Gregory Cauchie, and Michael Menth. Traffic Engineering for Bit Index Explicit Replication (BIER-TE). Internet-draft, Internet Engineering Task Force, July 2019. Work in Progress.
- [7] Collin Lee and John Ousterhout. Granular Computing. In *Proc. of ACM HotOS’19*, pages 149–154, Bertinoro, Italy, May 2019.
- [8] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. February 2019. arXiv: 1902.03383.
- [9] Xiaoye Steven Sun, Yiting Xia, Simbarashe Dzimamirira, Xin Sunny Huang, Dingming Wu, and TS Eugene Ng. Republic: Data multicast meets hybrid rack-level interconnections in data center. In *Proc. of IEEE ICNP’18*, pages 77–87, Cambridge, United Kingdom, September 2018.
- [10] Xiaoye Steven Sun and TS Eugene Ng. When creek meets river: Exploiting high-bandwidth circuit switch in scheduling multicast data. In *Proc. of IEEE ICNP’17*, pages 1–6, Toronto, Canada, October 2017.
- [11] O. Komolafe. Ip multicast in virtualized data centers: Challenges and opportunities. In *Proc. of IFIP/IEEE IM’17*, pages 407–413, Lisbon, Portugal, May 2017.
- [12] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. Passive realtime datacenter fault detection and localization. In *Proc. of USENIX NSDI’17*, pages 595–612, Boston, MA, March 2017.
- [13] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *Proc. of ACM CoNEXT’16*, pages 205–219, Irvine, CA, December 2016.
- [14] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *Proc. of IEEE MICRO’16*, pages 1–13, Taipei, Taiwan, October 2016.
- [15] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. In *Proc. of ACM SIGCOMM’16*, pages 525–538, Florianopolis, Brazil, August 2016.
- [16] Bill Fenner, Mark J. Handley, Hugh Holbrook, Isidor Kouvelas, Rishabh Parekh, Zhaojun (Jeffrey) Zhang, and Lianshu Zheng. Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised). RFC 7761.
- [17] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proc. of ACM SOSR’16*, pages 1–12, Santa Clara, CA, March 2016.
- [18] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. Riffa 2.1: A reusable integration framework for fpga accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 8(4), September 2015.
- [19] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. of ACM SIGCOMM’15*, pages 139–152, London, United Kingdom, August 2015.
- [20] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network’s (data-center) network. In *Proc. of ACM SIGCOMM’15*, pages 123–137, London, United Kingdom, August 2015.
- [21] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proc. of ACM SOSR’15*, pages 1–7, Santa Clara, CA, June 2015.

- [22] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Measuring control plane latency in sdn-enabled switches. In *Proc. of ACM SOSR’15*, pages 1–6, Santa Clara, CA, June 2015.
- [23] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.
- [24] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *Proc. of USENIX NSDI’15*, pages 117–130, Oakland, CA, May 2015.
- [25] Yiting Xia, TS Eugene Ng, and Xiaoye Steven Sun. Blast: Accelerating high-performance data analytics applications by optical multicast. In *Proc. of IEEE INFOCOM’15*, pages 1930–1938, Hong Kong, China, April 2015.
- [26] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, January 2015.
- [27] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proc. of ACM CoNEXT’14*, pages 75–88, Sydney, Australia, December 2014.
- [28] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proc. of USENIX OSDI’14*, page 583–598, Broomfield, CO, October 2014.
- [29] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro*, 34(5):32–41, September 2014.
- [30] Zhuhua Cai, Zekai J Gao, Shangyu Luo, Luis L Perez, Zografoula Vagena, and Christopher Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *Proc. of ACM SIGMOD’14*, pages 1371–1382, Snowbird, UT, June 2014.
- [31] Dan Li, Mingwei Xu, Ying Liu, Xia Xie, Yong Cui, Jingyi Wang, and Guihai Chen. Reliable multicast in data center networks. *IEEE Transactions on Computers*, 63(8):2011–2024, May 2014.
- [32] Xiaozhou Li and Michael J Freedman. Scaling IP multicast on datacenter topologies. In *Proc. of ACM CoNEXT’13*, pages 61–72, Santa Barbara, CA, December 2013.
- [33] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proc. of USENIX NSDI’12*, pages 253–266, San Jose, CA, April 2012.
- [34] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Proc. of USENIX NSDI’12*, pages 225–238, San Jose, CA, April 2012.
- [35] Dan Li, Henggang Cui, Yan Hu, Yong Xia, and Xin Wang. Scalable data center multicast using multi-class bloom filter. In *Proc. of IEEE ICNP’11*, pages 266–275, Vancouver, Canada, October 2011.
- [36] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *Proc. of ACM SIGCOMM’11*, page 98–109, Toronto, Canada, August 2011.
- [37] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *Proc. of ACM Workshop on Networking Meets Databases (NetDB’11)*, pages 1–7, Athens, Greece, June 2011.
- [38] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proc. of ACM IMC’10*, pages 267–280, Melbourne, Australia, November 2010.
- [39] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *Proc. of IEEE/IFIP DSN’10*, pages 527–536, Chicago, IL, June 2010.
- [40] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proc. of USENIX HotCloud’10*, pages 1–7, Boston, MA, June 2010.
- [41] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.
- [42] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VI2: A scalable and flexible data center network. In *Proc. of ACM SIGCOMM’09*, page 51–62, Barcelona, Spain, October 2009.

- [43] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. Lipsin: Line speed publish/subscribe inter-networking. In *Proc. of ACM SIGCOMM'09*, pages 195–206, Barcelona, Spain, August 2009.
- [44] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [45] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.
- [46] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.
- [47] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. of ACM SOSP'03*, pages 282–297, Bolton Landing, NY, October 2003.
- [48] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.
- [49] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: high-bandwidth multicast in cooperative environments. *ACM SIGOPS Operating Systems Review*, 37(5):298–313, 2003.
- [50] Bradley Cain, Steve E. Deering, Bill Fenner, Isidor Kouvelas, and Ajit Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376.
- [51] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proc. of ACM SIGCOMM'02*, pages 205–217, Pittsburgh, PA, August 2002.
- [52] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, January 2000.
- [53] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
- [54] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.
- [55] Multicast Command Reference for Cisco ASR 9000 Series Routers. <https://bit.ly/3AaVGDQ>. [Online; accessed February 2022].
- [56] NetFPGA SUME Reference Learning Switch Lite. <https://bit.ly/2UrUFlx>. [Online; accessed February 2022].
- [57] 10Gb Ethernet: The Foundation for Low-Latency, Real-Time Financial Services Applications and Other, Latency-Sensitive Applications. <https://bit.ly/33xtYST>. [Online; accessed February 2022].
- [58] Apache ActiveMQ. <http://activemq.apache.org>. [Online; accessed February 2022].
- [59] Apache Hadoop. <https://hadoop.apache.org>. [Online; accessed February 2022].
- [60] Apache openwhisk. <https://openwhisk.apache.org>. [Online; accessed February 2022].
- [61] Bess (berkeley extensible software switch). <https://github.com/NetSys/bess>. [Online; accessed February 2022].
- [62] Bringing Multicast to the Cloud. <https://bit.ly/3jP7naY>. [Online; accessed February 2022].
- [63] Data plan development kit (DPDK). <https://intel.ly/2GYxLAV>. [Online; accessed February 2022].
- [64] gRPC - An RPC library and framework. <https://github.com/grpc/grpc>. [Online; accessed February 2022].
- [65] Introducing data center fabric, the next-generation Facebook data center network. <https://bit.ly/3bWEDKG>. [Online; accessed February 2022].
- [66] Multicast group capacity: Extreme comes out on top. <https://bit.ly/2H5sQ1n>. [Online; accessed February 2022].
- [67] Open Config, Streaming Telemetry. <https://bit.ly/3kf7EEj>. [Online; accessed February 2022].
- [68] RabbitMQ. <http://www.rabbitmq.com>. [Online; accessed February 2022].
- [69] Using reliable multicast for data distribution with opendds. <https://bit.ly/3bWFefo>. [Online; accessed February 2022].
- [70] Why the world's largest hadoop installation may soon become the norm. <https://tek.io/33gDCsU>. [Online; accessed February 2022].
- [71] Xiaoye Sun. LDA Data Generator. https://github.com/sunxiaoye0116/data_generator/tree/dev. [Online; accessed February 2022].

Appendix A Supplementary Materials

This appendix includes materials that complement the contents presented in the paper.

A.1 Encoding Spine Downstream Links

The pseudo code of the ENCSPINEDSLINKS algorithm is shown in Algorithm 2. This algorithm encodes spine downstream links of the multicast tree into a label D and calculates the state \mathbb{S} to be maintained by spine switches.

A.2 Processing Spine Downstream Labels

The pseudo code of the PROCSPINEDSLABEL algorithm is shown in Algorithm 3. The algorithm processes two spine downstream labels: the common links among spine switches in the tree (denoted by \mathbb{C}), and the filter that encodes the remaining spine downstream links (denoted by D).

A.3 Overheads of Orca

Multicast offers significant bandwidth savings compared to unicast, and thus, it can scale data-intensive tasks that dominate datacenter networks. The authors of [36] reported that the communication time of data-intensive tasks using unicast can be larger than the computation time, especially as the number of workers increases. Achieving the benefits of multicast has been a long-standing problem. Orca achieves the benefits of multicast at the expense of the small overheads described below.

Server Resources. Orca agents require processing resources at servers. However, the computation performed on packets (mostly replacing labels) is quite simple and the memory needed to store leaf labels is small. Thus, Orca agents can easily be implemented on SmartNICs, which are getting popular in datacenters [3]. In this case, no CPU cores are taken away from the servers. Orca agents can also run on regular CPU cores. In this case, the agents consume only a small fraction of the available computing resources in each rack, as shown in the evaluation section. We note that since Orca is a multicast paradigm, the sender in the session transmits only one copy of each packet regardless of the number of receivers in the session. In contrast, in unicast, the sender needs to send a separate copy of each packet to every receiver, which for large-scale applications with many receivers and/or high data rates requires allocating additional CPU cores at the sender to sustain the needed data rate. That is, at the whole system level, the CPU resources used by Orca agents can be offset by the savings of CPU resources at the sender.

Packet Latency. Orca adds latency to packets at the leaf layer only, because the packets need to be sent to Orca agents for relabeling. This latency is in the order of one RTT within the rack, because of the simple processing done on packets by

Algorithm 2 Encode spine downstream links.

Input: \mathbb{T} : multicast tree
Input: \mathbb{C} : common ports in spine downstream switches
Input: F : filter size in bits
Output: D : computed spine downstream label
Output: \mathbb{S} : state sent to a subset of the spine switches

```
1: function ENCSPINEDSLINKS( $\mathbb{T}, \mathbb{C}, F$ )
2:   A Calculate a spine downstream label
3:    $D = \text{BitString}(\text{size} = F)$ 
4:   for ( $l \in \mathbb{T}.\text{spine\_ds\_links}()$ ) do
5:     if ( $l \notin \mathbb{C}$ ) then
6:        $D = D \cup \text{encode}(l.\text{id})$ 
7:   B Calculate false positive candidates
8:    $cands = \{\}$ 
9:   for ( $u \in \mathbb{T}.\text{spine\_switches}()$ ) do
10:    for ( $l \in u.\text{ds\_links}()$ ) do
11:      if ( $l \notin \mathbb{T}.\text{spine\_ds\_links}()$ ) then
12:         $cands = cands \cup (u, l.\text{dst})$ 
13:   C Calculate spine switch state
14:    $\mathbb{S} = \{\}$ 
15:   for ( $l \in cands$ ) do
16:     if ( $\text{check}(l.\text{id}, D)$  and  $l \notin \mathbb{C}$ ) then //false positive
17:        $\mathbb{S} = \mathbb{S} \cup \{(l.\text{src}, l.\text{id})\}$  // add link to state
18:   return  $\langle D, \mathbb{S} \rangle$ 
```

Orca agents. Most throughput-intensive datacenter applications, e.g., MapReduce [44], Hadoop [59], and Spark [40], can easily tolerate this small latency [33].

Communications Overheads. Orca achieves substantial bandwidth savings compared to the commonly-used unicast model. Orca, on the other hand, attaches a small, fixed-sized label (19 bytes) to each packet in typical datacenters; Orca label is smaller than the IP header. In addition, Orca uses additional bandwidth between leaf switches and agents deployed on servers within the same rack. Prior studies, however, reported that links at leaf layer are under-utilized. For instance, the study in [38] has found that the datacenter edge is lightly utilized: 80% of the time, the utilization is less than 10% for cloud and enterprise datacenters. A recent study by Facebook [20] reported that links between leaf switches and servers have a 1-minute average utilization of less than 1%.

A.4 Extensions and Limitations of Orca

Multipath Routing. In Orca, the multicast tree has one path from the source VM to any core switch, then it reaches the receivers by branching to spine and leaf switches. Orca can support multipath routing to achieve reliability and load balancing as follows. The centralized controller can compute multiple trees, each has the same source and receivers of the session but consists of different links. For example, the centralized controller can choose a different core switch as the root for each tree. It then calculates a different source label

Algorithm 3 Process a spine downstream label.

Input: D : spine downstream label
Input: l : downstream link attached to the spine switch
Input: \mathbb{C} : set of common ports in spine downstream switches
Input: $State$: state maintained at the spine switch
Output: **true** if duplicating a pkt on link l , else **false**

// Runs for every link attached to the spine switch

1: **function** PROCSPINEDSLABEL($D, l, State$)
 // Links belonging to \mathbb{C}
2: **If** $index(l.id) \in \mathbb{C}$ **then return true**
 // Checking the filter for $index(l.id) \notin \mathbb{C}$
3: **If not** $check(l.id, D)$ **then return false**
 // sID is the session ID included in the packet header
4: **return** $l.id \notin State[sID]$

for each tree, and instructs the source VM to store the new labels and spine switches to maintain state (if needed). The source attaches different source labels to the packets to instruct switches to forward them on links of different trees. Leaf labels are identical for all trees as they have the same receivers. As in other multipath routing systems, packet reordering may occur in this case and would need to be handled by the application.

Reliability and Congestion Control in Multicast. Prior works, e.g., [9, 31], proposed various methods for reliable transmission and congestion control for datacenter multicast. These methods can be used on top of Orca. In addition, the

Orca agent can reduce the number of control messages, e.g., ACK or NACK, since it can aggregate them per rack.

Incremental Deployment. Orca can run on legacy switches by encapsulating its labels in VLAN or VXLAN headers. The header identifier can be used to instruct switches to duplicate incoming packets.

Limitations of Orca. Deploying Orca in graph-based datacenter networks, e.g., Jellyfish [34] and Xpander [13], may require changes in some components of Orca. For example, although our server-assisted approach will work at the leaf layer in Jellyfish, Jellyfish's lack of structure does not allow Orca to use the same algorithms at other layers. A new label calculation algorithm would need to be designed to encode tree links without imposing assumptions on their layers.

A.5 Additional Simulation Results

We evaluate the running time of Orca and Elmo spent by the centralized controller when sessions change.

Running Time. Orca is simple and enables more updates per second to be processed by the control plane. For the Uni-S dataset, the average running time for Orca to calculate a session label is 0.34 ms (SD is 0.4 ms), while this average is up to 7.286 ms (SD is 9.8 ms) for Elmo-3. The average (SD) running times for Elmo-1, Elmo-2 and Elmo-4 are 6.1 ms (5.7 ms), 5.5 ms (5.6 ms) and 6.5 ms (8.6 ms), respectively. These times were measured on a workstation with a 2.3 GHz CPU.

Summary: *Orca calculates labels 21X faster than Elmo.*

Yeti: Stateless and Generalized Multicast Forwarding

Khaled Diab Mohamed Hefeeda

School of Computing Science

Simon Fraser University

Burnaby, BC, Canada

Abstract

Current multicast forwarding systems suffer from large state requirements at routers and high communication overheads. In addition, these systems do not support generalized multicast forwarding, where traffic needs to pass through traffic-engineered paths or requires service chaining. We propose a new system, called Yeti, to efficiently implement generalized multicast forwarding inside ISP networks and supports various forwarding requirements. Yeti completely eliminates the state at routers. Yeti consists of two components: centralized controller and packet processing algorithm. We propose an algorithm for the controller to create labels that represent generalized multicast graphs. The controller instructs an ingress router to attach the created labels to packets in the multicast session. We propose an efficient packet processing algorithm at routers to process labels of incoming packets and forwards them accordingly. We prove the correctness and efficiency of Yeti. In addition, we assess the performance of Yeti in a hardware testbed and using simulations. Our experimental results show that Yeti can efficiently support high speed links. Furthermore, we compare Yeti using real ISP topologies in simulations against the closest systems in the literature: a rule-based approach (built on top of OpenFlow) and two label-based systems. Our simulation results show substantial improvements compared to these systems. For example, Yeti reduces the label overhead by 65.3%, on average, compared to the closest label-based multicast approach in the literature.

1 Introduction

Recent large-scale Internet applications have introduced a renewed interest in scalable multicast services. Examples of such applications include live Internet broadcast (e.g., Facebook Live), IPTV [27], webinars and video conferencing [22], and massive multiplayer games [26]. The scale of these applications is unprecedented. For instance, due to the COVID-19 pandemic, a recent study [2] reported an increase by one order of magnitude within two months in video conferencing

traffic passing through a major European ISP. Moreover, Facebook Live aims to stream millions of live sessions to millions of concurrent users [8, 41]. To reduce the network load of such applications, ISPs can use multicast to efficiently carry the traffic through their networks. Examples of commercial systems using multicast include AT&T UVerse [40] and BT YouView [37]. Beyond multimedia systems, multicast is also useful for various applications such as real-time stock market updates, cloud applications [33], and publish-subscribe systems [24, 36, 39]. For instance, the CIO of the Japan Exchange Group highlighted the importance of multicast for their stock trading operations [32].

Large ISPs need to support *generalized* multicast forwarding to handle various business requirements. Specifically, providers of large-scale live applications require ISPs carrying their traffic to meet target quality metrics or SLAs (service level agreements), especially for popular/paid live multicast sessions. To meet the SLAs for various customers, ISPs may need to direct the traffic to network paths different from the minimum-cost ones computed by the routing protocols deployed in the ISP network. This is usually referred to as *traffic engineering*. Prior works, e.g., [7, 11, 12, 16], have proposed algorithms to support various traffic engineering objectives.

In addition, ISP customers may require their multicast traffic to pass though an *ordered* sequence of network services such as firewall, intrusion detection, and video transcoding before reaching the destinations. This is referred to as *service chaining*. Network services are usually deployed as virtual functions running on servers attached to *some* of the core routers in the ISP network. Previous works, e.g., [1, 5], presented algorithms for calculating optimal network paths to satisfy service chaining requirements.

Given the service chaining and traffic engineering requirements of recent applications, multicast sessions can no longer be represented as simple spanning trees. Rather, they need to be represented as general graphs. Efficiently forwarding traffic of multicast sessions represented as arbitrary graphs is, however, a challenging problem. One of the main concerns is the *state* that needs to be maintained at routers, which grows

linearly with the number of multicast sessions. This state also needs to be frequently updated to handle session changes and network dynamics, which imposes substantial communication and processing overheads, especially on core routers that need to support high-speed links carrying numerous sessions.

In this paper, we address the lack of scalable and generalized multicast forwarding systems for large-scale ISPs. In particular, we propose a *fully stateless* approach, called Yeti, to implement generalized multicast graphs. Yeti supports fast adaptation to network dynamics such as routers joining/leaving sessions and link failures, and it does not impose significant communication overheads. To the best of our knowledge, Yeti is the first multicast forwarding system that supports multicast sessions with traffic engineering and service chaining requirements. A high-level overview of Yeti is illustrated in Figure 1.

The main idea of Yeti is to completely move the forwarding information for each graph to the packets themselves as labels. Designing and processing such labels, however, pose key challenges that need to be addressed. First, we need to efficiently encode the graph forwarding information in as few labels as possible. Second, the processing overheads and hardware usage at routers need to be minimized. This is to support many concurrent multicast sessions, and to ensure the scalability of the data plane. Third, forwarding packets should not introduce *ambiguity* at routers. That is, while minimizing label redundancy and overheads, we must guarantee that routers will forward packets *on and only on* the links of the multicast graph. Yeti addresses these challenges.

This paper makes the following contributions:

- We propose a generalized multicast forwarding system that completely eliminates the state at routers; a long-standing problem for multicast. The proposed system supports service chaining and traffic engineering requirements.
- We design a control-plane algorithm to calculate an optimized label for each generalized multicast graph.
- We design an efficient packet processing algorithm for routers to handle labels attached to packets. The algorithm forwards packets only on links of the multicast graph. And it does not introduce any redundant traffic or create loops in the network.
- We present proofs to show the correctness of Yeti.
- We implement Yeti in a hardware testbed using a programmable router (NetFPGA) to demonstrate its practicality. Our results show that Yeti can support high-speed links carrying thousands of multicast sessions.
- We compare Yeti against a rule-based approach implemented using OpenFlow and the closest label-based approaches, LIPSIN [25] and BIER-TE [3], in simulations using real ISP topologies with different sizes. Our simulation results show that unlike Yeti which does not maintain state at any core router, the rule-based approach requires maintaining state at every router in the session

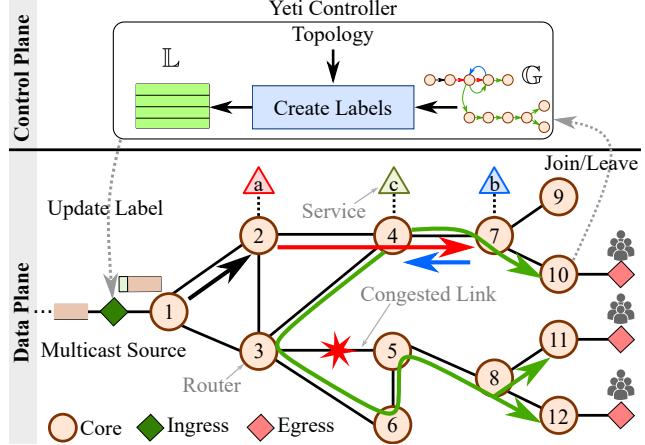


Figure 1: High-level overview of Yeti.

and LIPSIN maintains state at about 20% of the routers. In addition, Yeti reduces the label overhead by 65.3%, on average, compared to BIER-TE.

2 Related Work

We divide the related multicast forwarding works in the literature into stateful, stateless, and hybrid approaches.

Stateful Multicast Approaches. These multicast approaches require storing forwarding state about sessions at routers. The traditional IP multicast [31] is an example of such approaches. IP multicast is implemented in core routers produced by most vendors. However, it suffers from scalability issues in real deployments [29]. In particular, the group management and tree construction protocols, e.g., IGMP [28] and PIM [13], require maintaining state at routers for each session, and they generate control messages among routers to refresh and update this state. Thus, in practice, router manufacturers tend to limit the number of multicast sessions [38]. In addition, IP multicast uses shortest paths and cannot implement generalized graphs.

Recent SDN-based protocols, e.g., OpenFlow [17], can implement rule-based approaches, where a controller installs header-matching rules and actions to forward/duplicate packets. Since OpenFlow stores state at every router along the multicast trees, the total forwarding state at routers grows with the number of sessions.

Stateless Multicast Approaches. There are a few recent proposals for designing stateless multicast forwarding systems. For example, BIER [10] encodes global router IDs of tree receivers in the label as a bitmap. BIER supports only shortest paths and cannot realize traffic-engineered ones. A recent amendment to BIER, called BIER-TE [3], supports traffic-engineered trees. BIER-TE maps each bit position in the label to one of the links attached to routers in the network. It encodes the links of a multicast tree as corresponding bit positions in the calculated label. Upon receiving a packet,

a BIER-TE router checks the bit positions in the label. If the router matches one of its links in the label, it clears its position in the label and forwards/duplicates the packet on that link. The bitmap structure used in BIER-TE allows it to only implement multicast sessions represented as trees, and it cannot implement general multicast distribution graphs, as such graphs could have cycles to allow the multicast traffic to be processed by the specified set of network services. This is illustrated in the example multicast distribution graph in Figure 1, where packets of the session traverse the link between routers 4 and 7 three times to be processed by the services $a \rightarrow b \rightarrow c$.

Hybrid Multicast Approaches. Yeti is not the first system that moves forwarding information as labels attached to packets. However, prior systems did not support generalized forwarding, and they needed to maintain state at some or all routers belonging to the multicast tree. We refer to these systems as hybrid approaches. For example, the early work by Chen et al. [30] proposed a label-based system that attaches link IDs to every packet in a multicast session, and removes unwanted portions of the label as the packet traverses the network. The processing algorithm in [30] requires maintaining state at every router belonging to a multicast session in order to remove the labels, which is not scalable. Later works, e.g., mLDP [23], enable multicast in label-switched paths (LSPs). mLDP forwards traffic on the shortest paths and thus cannot support traffic-engineered trees. It also requires an additional protocol to distribute labels among routers.

LIPSIN [25] uses a Bloom filter as label to encode global link IDs of a tree. LIPSIN may result in redundant traffic or forwarding loops, because of the probabilistic nature of Bloom filters. Thus, LIPSIN requires an additional protocol where downstream routers notify upstream ones if they falsely receive a packet. This protocol imposes additional state and communication overheads on routers.

Segment routing (SR) [4] is a recent proposal to support traffic-engineered unicast flows. It was later extended to support multicast by considering every tree as a segment in the network. It attaches one label containing the tree ID to packets of a session. Routers along the tree maintain a mapping between that label and the output interfaces. That is, the SR multicast extensions require maintaining state at routers for every tree.

In §5, we compare Yeti versus a rule-based approach implemented in OpenFlow, LIPSIN, and BIER-TE as they are the closest stateful, hybrid, and stateless multicast systems, respectively, that can support traffic engineering requirements.

3 Problem Definition and Solution

We start this section by specifying the considered problem and its challenges. We next describe an overview of Yeti and its main components. This is followed by the details of each

component. In the Appendix §C, we present an illustrative example of Yeti to demonstrate its details.

3.1 Problem Definition and Challenges

The problem considered in this paper is how to efficiently forward the traffic of a *generalized* multicast session that may need to be processed by an ordered set of network services and/or directed through a specific set of network paths within the ISP network.

For illustration, consider the ISP network in the lower part of Figure 1, which contains ingress, core, and egress routers marked by different shapes and colors. Some of the core routers are connected to servers offering various (virtualized) network services such as intrusion detection and video transcoding. There is a multicast source connected to the ingress router and multiple receivers reachable through the three egress routers. The creator of the multicast session requires the traffic to be processed by the three network services $a \rightarrow b \rightarrow c$ in order. In addition, the ISP implements traffic engineering mechanisms for various objectives, e.g., to minimize the maximum link utilization (MLU), which requires the traffic to avoid the link $3 \rightarrow 5$ in this example. The colored arrows in the figure show the different paths taken by the traffic of the multicast session to reach all receivers. These paths form a graph (not tree), which we refer to as the *multicast distribution graph* \mathbb{G} . Note that nodes in the distribution graph represent routers, not end users. The top right part of Figure 1 shows the graph for the multicast session marked in the lower part of the figure.

Our problem then becomes how to get routers in the ISP to forward the traffic of a multicast session represented by an arbitrary multicast distribution graph \mathbb{G} . Existing algorithms in the literature, e.g., [1, 12], can be used to calculate \mathbb{G} to satisfy various service chaining and traffic engineering requirements; our proposed (forwarding) solution is orthogonal to the calculation of \mathbb{G} and can work with any of them.

The arbitrary nature of the distribution graph makes designing scalable multicast forwarding systems a challenging problem. A possible approach to address this problem is to maintain state (e.g., in the form of match-action rules) at routers. However, as mentioned in §2, maintaining state at routers is not scalable even for traditional multicast forwarding without service chaining and traffic engineering requirements. This is because the state, which grows linearly with the number of multicast sessions, not only consumes the scarce SRAM resources of routers, but it also needs to be frequently updated to handle network failures and session dynamics (e.g., router joining/leaving). The cost of updating the state consists of two factors. First, routers need to process many update (control) messages while processing data packets, which may result in slowing down the data plane operations. Second, frequent state updates negatively impact the network agility and consistency, because the control plane has to *schedule* the

updates to corresponding routers to ensure consistency [20], since greedy state updates may result in violating the SLA objectives [6].

To reduce state, a part or all of the forwarding information can be moved to labels which are attached to the packets of multicast sessions, where routers use these labels in the forwarding decisions. However, efficiently representing multicast graphs in compact labels is difficult, especially for multicast sessions that have service chaining and traffic engineering requirements. If not carefully designed, labels representing a multicast graph can grow large in size and hence impose significant communication overheads, and more critically they could introduce *ambiguity* at routers, i.e., routers may not be able to decide which interface(s) to forward the packets on. This may introduce duplicate packets and forwarding loops, which substantially increases the load on the ISP network and wastes its bandwidth and processing resources.

In summary, forwarding generalized multicast graphs presents multiple challenges that Yeti addresses. Specifically, stateful approaches *reduce scalability* as they impose substantial memory and processing overheads on switches. On the hand, current stateless approaches significantly *increases packet sizes* and may *introduce forwarding ambiguity*. This would defeat the main purpose of deploying multicast in the first place. Yeti breaks this long-standing trade-off between scalability, efficiency, and correctness by completely moving the forwarding state into compact labels, and carefully processing them in the data plane.

3.2 Solution Overview

Yeti is a *stateless* multicast forwarding system that efficiently implements general multicast graphs inside a single ISP network; extending Yeti to support multiple ISPs is described in §3.6. As shown in Figure 1, the ISP network has data and control planes. The data plane is composed of routers. Every router is assigned a unique ID, and it maintains two data structures: Forwarding Information Base (FIB) and interface list. FIB provides reachability information between routers using shortest paths. The interface list maintains the IDs of all local interfaces. The control plane (or the *controller*) learns the ISP topology, shortest paths between routers, and interface IDs for every router, which is done using common intra-domain routing and monitoring protocols.

Yeti consists of a centralized controller and a packet processing algorithm. The controller calculates the distribution graph for a multicast session using existing algorithms, e.g., [1, 12], and it creates, using our algorithm in §3.4, an optimized set of labels \mathbb{L} to realize this graph in the data plane. As detailed in §3.3, Yeti defines four types of labels; each serves a specific purpose in encoding the graph efficiently. The controller sends the set of labels \mathbb{L} to the ingress router of the session, which in turn attaches them to all packets of the session. When a graph G changes (due to link failure

Name	Type	Content	Content Size (bits)
FSP	00	Global router ID	$1 + \lceil \log_2 N \rceil$
FTE	01	Local interface ID	$\lceil \log_2 I \rceil$
MCT	10	Interface bitmap	I
CPY	11	Label range (in bits)	$\lceil \log_2 (N \times \text{size(FTE)}) \rceil$

Table 1: Label types in Yeti. N is the number of routers, and I is the maximum number of interfaces per router.

or egress router joining/leaving the session), the controller creates a new set of labels and sends them *only* to the ingress router, no other routers need to be updated.

The packet processing algorithm, described in §3.5, is deployed at core routers. It processes the labels attached to packets and forwards/duplicates packets based on these labels. It also determines the subset of labels to attach to the forwarded packets such that the subsequent routers can realize the distribution graph without any ambiguity, forwarding loops, or redundant traffic. We present a theorem proving the correctness of Yeti in §3.6.

We present an illustrative example in the **Appendix**. This example implements the distribution tree in Figure 1, and it shows the details of creating labels at the controller and processing packets at routers.

3.3 Label Types in Yeti

Yeti is a label-based system. Thus, one of the most important issues is to define the types and structures of labels in order to minimize the communication and processing overheads, while still being able to represent generalized multicast graphs. We propose the four label types shown in Table 1. The first two label types are called *Forward Shortest Path* (FSP) and *Forward Traffic Engineered* (FTE). They are used by routers to forward packets on paths that have no branches. The other two label types are *Multicast* (MCT) and *Copy* (CPY). The MCT label instructs routers to duplicate a packet on multiple interfaces, and the CPY label copies a subset of the labels. Every label consists of two fields: *type* and *content*. The *type* field is used by routers to distinguish between labels during parsing, and the *content* field contains the information that this label carries. The size of a label depends on the size of the *content* field.

We use the example topology in Figure 2 to illustrate the rationale used in defining Yeti labels. In the figure, solid lines denote tree links and the dotted line denotes a link on the shortest path to some destinations. The ISP avoids it because it is congested in this example.

We divide a multicast graph into *path segments* and *branching points*. A path segment is a contiguous sequence of routers without branches. If a path satisfies any sub-sequence of the service chaining requirements of a session, the path segment

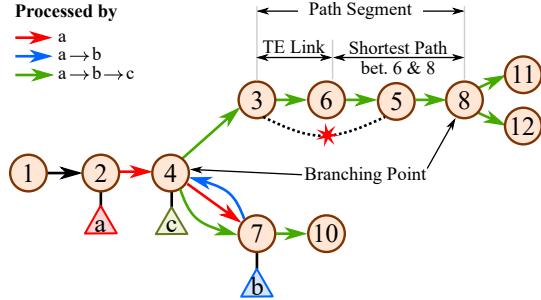


Figure 2: Illustration of path segments and branching points in Yeti. Segment $3 \rightarrow 8$ does not follow the shortest path.

ends when there is a router with at least one service. A branching point refers to a router that duplicates packets on multiple interfaces. For the example in Figure 2, there are five path segments: $\{1 \rightarrow 2\}$, $\{2 \rightarrow 4 \rightarrow 7\}$, $\{7 \rightarrow 4\}$, $\{7 \rightarrow 10\}$, and $\{3 \rightarrow 6 \rightarrow 5 \rightarrow 8\}$. Routers 4 and 8 are branching points.

The basic label in Yeti is FTE, where a router is instructed to forward the packet carrying the label on a specific interface. In many situations, however, packets follow a sequence of routers on the shortest path. For these situations, we define the FSP label, which replaces multiple FTE labels with just one FSP label. An FSP label contains a global router ID, which instructs routers to forward incoming packets on the shortest path to that router. In addition, the first bit in an FSP label indicates whether the packet will be processed at the corresponding router. For example, in Figure 2, instead of using two FTE labels for the links $\{6 \rightarrow 5\}$ and $\{5 \rightarrow 8\}$, Yeti uses one FSP label with destination ID set to node 8. In large topologies, FSP labels significantly reduces label overheads.

FTE and FSP labels can represent path segments, but they cannot handle branching points where packets need to be duplicated on multiple interfaces. Notice that, after a branching point, each branch needs a different set of labels because packets will traverse different routers. To handle branching points, we introduce the MCT and CPY labels. The MCT label instructs routers to duplicate packets on multiple interfaces using a bitmap of size I bits, where I is the maximum interface count per router. The bitmap represents local interface IDs, where the bit locations of the interfaces that the packet should be forwarded on are set to one. The CPY label does not represent a forwarding rule. Instead, it instructs a router to copy a subset of labels when duplicating packets to a branch without copying all labels. Specifically, consider a router that duplicates packets to n branches. The MCT label is followed by n sets of labels to steer traffic in these branches, where every label set starts with a CPY label. The CPY label of one branch contains an offset of label sizes (in bits) to be duplicated to that branch. For example, in Figure 2, router 4 will process an MCT label followed by two CPY labels for the traffic represented with a green arrow, one for each of the two branches. The CPY label content size in Table 1

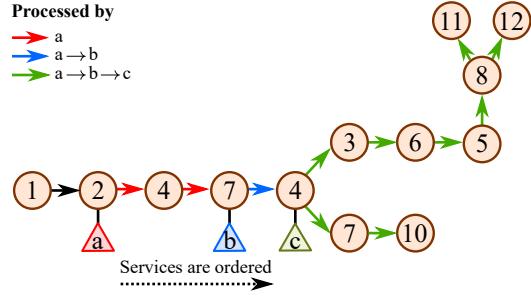


Figure 3: The resulting tree \mathbb{T} for the graph \mathbb{G} in Figure 2.

uses the worst-case scenario. This happens when the graph has the largest diameter, which is $O(N)$ and every link is traffic-engineered, where N is the number of core routers.

3.4 Creating Yeti Labels at the Controller

The ENCODEGRAPH algorithm, shown in Algorithm 1, runs at the controller to create labels. We omit the pseudo code for some functions that the algorithm calls due to space limitations. When the distribution graph \mathbb{G} changes, the algorithm calls the BUILDTREE function to create a tree \mathbb{T} that reflects the order of network services needed before reaching the destinations. Then, the ENCODEGRAPH algorithm calls the CREATELABELS function with the created tree to calculate a new set of labels \mathbb{L} to encode the graph paths and sends them to the ingress router, which attaches them to every packet in the session. The details of our algorithms are as follow.

Building the Tree. The BUILDTREE function traverses the multicast graph and creates a list of tuples for every path and provided services on that path. For example, in Figure 2, the tuple $\{\{7 \rightarrow 4\}, \{a \rightarrow b\}\}$ represents packets traversing the path $\{7 \rightarrow 4\}$ after processed by the services a and b.

The BUILDTREE function then traverses these tuples from the tuple starting with the source node. The function keeps track of the current parent and visited nodes in the tree, and builds the tree \mathbb{T} as follows. For every tuple, the function creates nodes with every router ID and the provided services in that tuple. For the tuple mentioned earlier, the function creates the nodes: $(7, \{a \rightarrow b\})$ and $(4, \{a \rightarrow b\})$. The function adds a node to \mathbb{T} if it did not exist before. Then, the function adds that node to the children of current parent, and sets the new node as the current parent. If a node exists in the tree, the function sets it as the current parent. Figure 3 depicts the resulting tree of the graph in Figure 2.

Creating Labels. The CREATELABELS algorithm divides \mathbb{T} into segments and branching points. The algorithm calculates FSP and FTE labels for every segment, and MCT and CPY labels at branching points. The label order is important because it reflects which routers process what labels. The algorithm traverses the core routers of \mathbb{T} in a depth-first search order starting from the core router connected to the ingress router.

Algorithm 1 Encode a multicast graph into labels.

Input: \mathbb{G} : multicast graph
Input: S : ordered list of services in the session
Input: \mathbb{P} : shortest path map
Output: \mathbb{L} : labels to be sent to the ingress router

```
1: function ENCODEGRAPH( $\mathbb{G}, S, \mathbb{P}$ )
2:    $\mathbb{T} = \text{BUILDTREE}(\mathbb{G}, S)$ 
3:   return CREATELABELS( $\mathbb{G}.src, \mathbb{T}, S, \mathbb{P}$ )
4: function CREATELABELS( $source, \mathbb{T}, S, \mathbb{P}$ )
5:    $\mathbb{L} = \{\}, pth\_seg = \{\}, stack = \{source\}$ 
6:   while ( $stack.size() > 0$ ) do
7:      $r = stack.pop() // a router in \mathbb{T}$ 
8:     // Services provided by r for the session
9:      $srv = S.at(r)$ 
10:     $core\_children = \mathbb{T}.core\_children(r) // core routers$ 
11:     $children = \mathbb{T}.children(r) // core and egress routers$ 
12:    // Build a path segment pth_seg
13:    if ( $core\_children.size() == 1$ ) then
14:      |  $pth\_seg.append(r); stack.push(children[0])$ 
15:    // Handle a path segment (create FSP & FTE)
16:    // S[0] is the next expected service
17:    if ( $core\_children.size() == 0$  or  $S[0] \in srv$ ) then
18:      |  $pth\_seg.append(r)$ 
19:      |  $lbls = \text{CALCSEGMENTLBL}(\mathbb{T}, pth\_seg, \mathbb{P})$ 
20:      |  $\mathbb{L}.push(lbls); pth\_seg = \{\}$ 
21:      |  $S.remove(srv)$ 
22:    // Handle branching point (create MCT & CPY)
23:    else if ( $children.size() > 1$ ) then
24:      | if ( $pth\_seg.size() > 0$ ) then
25:        | |  $pth\_seg.append(r)$ 
26:        | |  $lbls = \text{CALCSEGMENTLBL}(\mathbb{T}, pth\_seg, \mathbb{P})$ 
27:        | |  $\mathbb{L}.push(lbls); pth\_seg = \{\}$ 
28:        | |  $\langle mct\_lbl, cpy \rangle = \text{CREATEMCT}(children)$ 
29:        | |  $\mathbb{L}.push(mct\_lbl)$ 
30:        | | if ( $cpy$ ) then // Creating CPY labels
31:          | | | for ( $c \in children$ ) do
32:            | | | | // A recursive call for each branch
33:            | | | |  $br\_lbls = \text{CREATELABELS}(c, \mathbb{T}, S, \mathbb{P})$ 
34:            | | | |  $offset = \text{CALCLABELSIZE}(br\_lbls)$ 
35:            | | | |  $\mathbb{L}.push(CPY(offset)); \mathbb{L}.push(br\_lbls)$ 
36:    return  $\mathbb{L}$ 
```

It keeps track of the router r that is being visited, and one path segment (pth_seg). Once a router r is visited, if r has only one core child (Line 13 in Algorithm 1), this means that r belongs to the current segment. The algorithm then appends r to pth_seg , and pushes its child to the stack to be traversed later. For example, the algorithm pushes routers 3, 6, 5 and 8 in Figure 3 to pth_seg because each of them has only one child. If r has no core children or it provides some services (has a path segment), or r has more than one child (has a branching point), the algorithm calculates labels as follows.

Handling Path Segments. The CREATELABELS algorithm creates a label for a path segment when pth_seg ends. This happens in three cases. First, when r is connected to an egress router (e.g., router 10 in Figure 3). Second, when r is a branching point and pth_seg is not empty (e.g., router 8 in Figure 3). Third, when r provides at least one service (e.g., router 2 in Figure 3). In all cases, the algorithm appends r to pth_seg and calculates FSP and FTE labels using CALCSEGMENTLBL.

CALCSEGMENTLBL takes as inputs a tree \mathbb{T} , a path segment pth_seg and the shortest path map \mathbb{P} , and calculates the FSP and FTE labels of the given pth_seg . It divides pth_seg into two sub-paths: one that follows the shortest path, and one that does not. It then recursively applies the same to the latter sub-path. Specifically, CALCSEGMENTLBL concurrently traverses pth_seg and the shortest path between source and destination. It stops when the traversal reaches a router in pth_seg that does not exist in the shortest path. This means that this router does not follow the shortest path, hence, it adds an FSP label for the previous router. If pth_seg has routers that do not follow the shortest path, CALCSEGMENTLBL adds an FTE label and recursively calls itself using a subset of pth_seg that is not traversed so far. CALCSEGMENTLBL does not generate two consecutive FSP labels. When it calculates an FSP label, it either terminates, or creates an FTE label followed by a recursive call.

For the example in Figure 3, the CALCSEGMENTLBL algorithm processes the segment $\{3 \rightarrow 6 \rightarrow 5 \rightarrow 8\}$ as follows. It finds that the link $(3, 6)$ is not on the shortest path from 3 to 8. It calculates an FTE label for this link, and recursively calls itself with the sub-path $\{6 \rightarrow 5 \rightarrow 8\}$ as an input, for which the algorithm creates an FSP label with router ID 8.

Handling Branching Points. The CREATELABELS algorithm calculates MCT and CPY labels at branching points. The algorithm calls CREATEMCT that returns MCT label and a boolean value cpy indicating whether CPY labels are required. To create an MCT label, CREATEMCT initializes an empty bitmap of width $I + 1$ (I is the maximum interface count per router). For every child c of r , it sets the bit location in this bitmap that represents the interface ID between r and c . It checks if CPY labels are needed as follows. If any child c has at least one core child, this means that this core child needs labels to forward/duplicate packets. Otherwise, if all children have no other core children, the router r is either directly connected to an egress router, or its children are connected to egress routers. Thus, these routers do not need more labels and Yeti does not create CPY labels for these branches. For example, at router 4 in Figure 3, core children 3 and 7 have other core children which are 6 and 10, respectively. Hence, two CPY labels are created for the two branches at 4. The algorithm does not create CPY labels at router 8, because its core children 11 and 12 have no other core children.

Recall that a CPY label copies a subset of labels at a specific branch. If CPY labels are needed at the branching point and r has n children/branches, the MCT label is followed by

n CPY labels, and every CPY label is followed by labels to forward packets on the corresponding branch. Specifically, the algorithm iterates over the children of r . For every child c , the algorithm adds an FSP label if the child provides services. Then, the algorithm recursively calls `CREATELABELS` to create labels of the corresponding branch (Line 33). The created CPY label for a branch contains the size of this branch labels in bits to be copied. We calculate this size by accumulating the size of every label type in br_lbls (Line 34).

Time and Space Complexities. In the worst-case scenario, when every router is a branching point, the `ENCODEGRAPH` algorithm needs to create labels for each branch. Thus, the time complexity of the `ENCODEGRAPH` algorithm is $O(K^2N^2 + M)$, where N is the number of routers, M is the number of links, and K is the maximum service chain length. We note that the values of N , M and K are not large for realistic ISP networks. The number of ISP routers N is in the range of 10’s–100’s [9, 18], most ISP networks are sparse with number of links M ranging from 500 to around 2,000, and the length of service chains K ranges from 2–10 [1]. Given these practical values, the `ENCODEGRAPH` algorithm can easily run on a commodity server. Notice that the `CALCSEGMENTLBL` algorithm processes the segment after all routers of that segment is traversed. Thus, it only adds linear overhead to the first term of the time complexity. The space complexity of the `ENCODEGRAPH` algorithm is $O(N^2D)$, where D is the diameter of the network.

3.5 Processing Yeti Packets

The proposed packet processing algorithm is to be deployed at core routers, and it processes Yeti packets. This is done by setting a different Ethernet type for Yeti packets at ingress routers. A core router checks the Ethernet type of incoming packets, and invokes the processing algorithm if they are Yeti packets. The algorithm forwards/duplicates packets and it removes labels that are not needed by next routers. It also copies a subset of labels at branching points.

The packet processing algorithm works as follows. If the packet has no labels, this means the packet reached a core router that is attached to an egress router. So, the packet is forwarded to that egress router. Otherwise, the algorithm processes labels according to the following cases:

(1) *FSP Label.* If the FSP label content is not the receiving router ID, it means that this router belongs to a path segment. The algorithm then forwards the packet along the shortest path based on the underlying intra-domain routing protocol *without* removing the label. If the FSP content equals the router ID, this means that the path segment ends at this router. The algorithm first checks whether the packet needs to be processed by services connected to that router. If the first bit is set, then the packet is forwarded to the datacenter. Otherwise, the algorithm removes the current label and calls the packet processing algorithm again to process next labels. This is

because the packet may have other labels.

(2) *FTE Label.* The algorithm removes the label, extracts the local interface ID, and forwards the packet on that interface.

(3) *MCT Label.* The algorithm first copies the original labels, and removes the labels from the packet. It then extracts the MCT content into mct . The MCT label contains the interface ID bitmap ($mct.intfs$) and whether it is followed by CPY labels ($mct.has_cpy$). The algorithm iterates over the router interfaces in ascending order of their IDs. It locates the interfaces to duplicate the packet on. For every interface included in the MCT label, the algorithm clones the packet. If the MCT label is followed by CPY labels, the algorithm removes the corresponding CPY label, extracts the following labels based on the offset value, and forwards the cloned packet on the corresponding interface.

Time and Space Complexities. The time complexity of the algorithm is $O(I)$, where I is the maximum interface count per router. The algorithm does not require additional space at routers.

3.6 Analysis and Practical Considerations

Analysis. The following theorem proves the correctness of Yeti. That is, Yeti forwards packets on and only on links belonging to the multicast graph. This is a critical objective for large-scale multicast sessions, as redundant traffic wastes network resources and overloads routers. Due to space limitation, we show the full proof in the Appendix §A.

Theorem 1 (Correctness). *Yeti forwards packets on and only on links that belong to the multicast graph.*

Proof Sketch. Yeti guarantees correctness by creating an ordered set of labels to realize the given multicast distribution graph. We analyze the order and type of the created labels and prove that they do not result in forwarding loops or redundant traffic while delivering the traffic to all receivers in the multicast session. □

Practical Considerations. The description of Yeti has focused on offering a scalable multicast service within a *single* ISP using programmable routers. In the Appendix §B, we describe simple techniques to enable Yeti across multiple ISPs and its incremental deployment.

4 Evaluation in a Testbed

We present a *proof-of-concept* implementation of the proposed multicast forwarding system, and we conduct experiments in a testbed with a NetFPGA programmable router.

In addition, since switches that support the P4 programming language [21], e.g., Tofino, are getting popular in practice, we also implemented Yeti in P4. We obtained a license for the Intel P4 software development environment (SDE) version 9.5.0, which contains various tools, including a P4

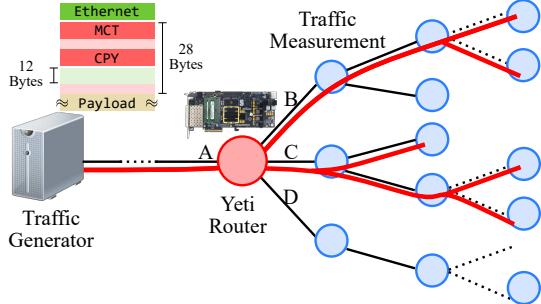


Figure 4: Setup of our testbed.

compiler (`bf-p4c`) and a switch model. The compiler produces code that runs on Tofino switches. We validated our implementation using the switch model included in the SDE. The details of the P4 implementation of Yeti are presented in Appendix §D.

4.1 Testbed Setup

The testbed, shown in Figure 4, has a Yeti Router representing a core router in an ISP topology that receives and processes packets of concurrent multicast sessions. We implemented the Yeti Router in a programmable processing pipeline using NetFPGA SUME [19], which has four 10GbE ports. The testbed also has a 40-core server with an Intel X520-DA2 2x10GbE NIC, which is used to generate traffic of multicast sessions at high rate using MoonGen [14].

Our router implementation is based on the open source project in [34]. This project contains three main Verilog modules: `input_arbiter`, `output_port_lookup` and `output_queues`. Our implementation modifies the last two modules in the router as follows. The `output_port_lookup` module is modified to read the first label to decide which ports to forward/duplicate the packet on. For each duplicated packet, it decides the labels to be detached and maintains this information in the packet metadata. We also modified the `output_queues` module to run at every output queue of the router, and detach labels that are not needed in the outgoing packets.

4.2 Experiments and Results

We transmit labelled packets of concurrent multicast sessions at the maximum link speed in our testbed (10 Gbps) from the traffic-generating server to the Yeti Router. We stress Yeti by transmitting traffic that requires copying and rearranging labels for each packet. In every experiment, we attach labels with different sizes to each packet. These labels contain MCT and CPY labels. The MCT label instructs the Yeti Router to duplicate packets on two ports B and C in Figure 4. We report the results for a sample label size of 28 bytes. This is because, as we show in §5, most of the packets have this label size for

Latency (μsec)	50 th %	95 th %	99.9 th %
Yeti	960.4	973	1,042
MAC forwarding	960.3	972.9	1,040
Difference	0.1	0.1	2

Table 2: Packet latency (in μ sec) measured in our testbed.

traffic engineering and service chaining scenarios in different ISP networks. The CPY labels instruct Yeti to copy 12 and 16 bytes to ports B and C, respectively. We measure the outgoing traffic on port B. The main parameter that we control is the packet size, which we vary from 64 to 1024 bytes. We report three important metrics for the design of high-end routers: packet latency, resource usage, and throughput.

Latency. We report the packet processing latency at port B in Figure 4 when the Yeti Router processes CPY labels (i.e., the worst-case scenario in terms of processing). We measure the latency by timestamping each packet at the traffic generator, and taking the difference between that timestamp and the time the packet is received at port B. We use the Berkeley Extensible Software Switch [15] to timestamp packets. Since it may add overheads while timestamping and transmitting packets, we compare the latency of Yeti processing against the basic forwarding in the same testbed, which is done by matching the fixed-length MAC address. Table 2 shows multiple statistics of the packet latency for both Yeti and unicast forwarding when the packet size is 1,024 bytes. The table shows that the latency of Yeti processing under stress is close to the simple unicast forwarding. For example, the difference of the 95th percentiles of packet latency is only 0.1 μ sec when the packet size is 1,024 bytes.

Resource Usage. We measure the resource usage of the packet processing algorithm, in terms of the number of used look-up tables (LUTs) and registers in the Yeti Router, which are generated by the Xilinx Vivado tool after synthesizing and implementing the project. Our implementation uses 12,677 slice LUTs and 1,701 slice registers per port. Relative to the available resources, the used resources are only 3% and 0.2% of the available LUTs and registers, respectively. Thus, Yeti requires small amount of resources while it can forward traffic of many concurrent multicast sessions.

Throughput. In Figure 5, we compare the rate of incoming packets to the Yeti Router versus the rate of packets observed at port B. The figure shows that the numbers of transmitted and received packets per second are the same (i.e., no packet losses). The figure also shows that our algorithm can sustain the required 10 Gbps throughput for all packet sizes.

We realize that core routers have large port density and high speeds. We believe that Yeti can achieve line-rate performance in these routers, because Yeti processes each incoming packet independently and adds small processing latency per packet as shown in Table 2.

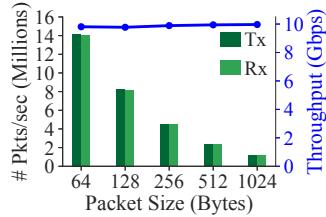


Figure 5: Throughput of received traffic from our testbed.

5 Evaluation using Simulation

We analyze the performance of Yeti and compare it against the closest multicast approaches using simulation.

5.1 Simulation Setup

Simulator. We implemented a Python-based simulator to compare the performance of different multicast systems in large setups using realistic ISP topologies. The simulator has two components. The first acts as the Yeti controller in Figure 1. When this component receives an egress router event, it updates the corresponding multicast graph, and then generates labels using Algorithm 1. The second component simulates the packet processing algorithm in §3.5. The simulator also implements prior systems for comparisons.

ISP Topologies. We use 14 topologies in the Internet Topology Zoo dataset [35]. This dataset represents a wide range of actual ISPs, where the number of routers ranges from 36 to 197, and the number of links ranges from 152 to 486.

Multicast Sessions. We simulate dynamic and diverse multicast sessions. The source of each session is randomly selected from one of the ISP routers. The session bandwidth is randomly chosen from the set $\{0.5, 1, 2, 5, 10\}$ Mbps, which represents the bandwidth values of different types of applications. The session duration is randomly assigned to a value from $\{10, 20, 40, 60, 80, 100, 120\}$ minutes. These values reflect a wide range of short to long multicast sessions. In addition, while the session is active, we make its receivers join and leave according to random events generated from a Poisson distribution, where 60% of these are join events and 40% are leave events. We make the receiver join rate 50% higher than the leave rate to incrementally stress the system with more multicast receivers as the time passes.

Each multicast session requires a set of network services, or service chain. We vary the length of the service chain from 3 to 5 as these lengths represent common service usage patterns [1, 42]. To represent practical deployment of services in ISPs, we divide services to essential and supplementary according to their popularity [42]. Essential services such as firewalls are deployed at all ISP locations, whereas supplementary services such as video encoders are only deployed at some of the ISP locations. To stress our system, we set the percentage of ISP locations that provide supplementary

services to only 25%. Each multicast session includes two randomly chosen essential services and the rest of the services are supplementary ones, also randomly chosen.

Since Yeti does not dictate how multicast graphs are computed, we use the algorithms in [12] and [1] to calculate the graphs based on the traffic engineering and service chaining requirements, respectively.

Experiments and Statistics. We simulate the operation of an ISP managing concurrent and dynamic multicast sessions over an extended period of time (24 hours), where about 200,000 sessions are created over the simulation period. Specifically, we first choose one of the 14 ISP topologies and generate the multicast sessions using the characteristics described above. Then, we repeat the experiment for the same ISP topology five times, starting from different seeds for the random distributions. Thus, for each ISP topology, we collect and analyze statistics from about 1M randomly generated, diverse, and dynamic multicast sessions. Then, the whole process is repeated for each of the 14 ISP topologies.

We report the 95-percentile of various performance metrics in the following subsections, as it reflects the performance over extended number of sessions. We present representative samples of our figures, using the ISP topologies with the largest and median numbers of routers. We also present averages and normalized averages (per router) across all ISP topologies, to infer the performance in general settings. When we present the (normalized) averages, we report the standard deviation in each case preceded by \pm .

Yeti is the first multicast forwarding system to support service chaining and traffic engineering. Thus, we first compare a simpler version of Yeti against the state-of-art systems for multicast sessions with traffic engineering requirements as these cannot support service chaining. Then, we analyze the performance of Yeti for multicast sessions with traffic engineering and service chaining requirements.

5.2 Yeti vs Stateful and Hybrid Approaches

We compare Yeti versus the closest stateful and hybrid multicast forwarding systems, which are OpenFlow [17] and LIPSIN [25]. We implemented a rule-based multicast forwarding system using OpenFlow [17] (referred to as RB-OF), because rule-based is a general packet processing model that is supported in many networks. The rule-based system is stateful as it installs match-action rules in routers.

LIPSIN is a hybrid approach that encodes the tree link IDs of a session using a Bloom filter. For every link, the LIPSIN controller maintains D link ID tables with different hash values. LIPSIN creates the final label by selecting the table that results in the minimum false positive rate. Since LIPSIN may result in false positives, each router maintains state about incoming links and the label that may result in loops for every session passing through this router. We set the filter size of LIPSIN to the 99th-percentile of the label

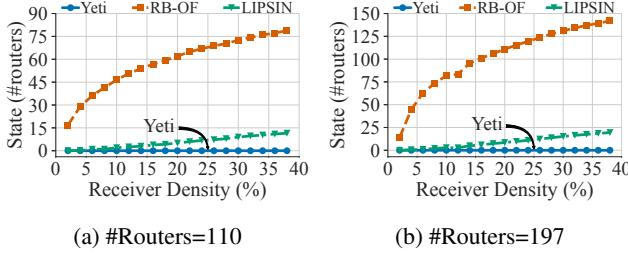


Figure 6: State size.

size of Yeti. This enables LIPSIN to encode a large number of links per tree in its labels. We use the same parameters proposed for LIPSIN: we set D to 8 tables and use five hash functions per link. We use Bloom filters and Murmurhash3 hashing functions.

State Size. Figure 6 shows the 95-percentile of the state size per multicast session as the density of receivers varies, which corresponds to the number of core routers needed to maintain state. The results are shown for the median and largest topologies with sizes 110 and 197 routers, respectively. The results for other topologies are similar (shown in Appendix §E).

First, notice that Yeti does not require any state at any core router. In contrast, RB-OF needs to maintain state at each router and that state increases with the topology size as well as the density of receivers in each multicast session. For example, the state size increases from 80 to 130 rules when the receiver density increases from 10% to 30%. LIPSIN, on the other hand, needs to maintain state at up to 20 routers when the receiver density is 40%. In this case, multicast graphs span 50% of the routers. That is, LIPSIN needs to maintain state at up to 20% of the routers in the multicast graph.

State Update Rate. Maintaining state at routers does not only consume their limited SRAM, but also increases the overheads of updating this state at routers when the distribution graph changes [6, 20].

We assess the average number and percentage of routers to be updated when a single multicast tree changes, and show the results for sample ISP topologies with various sizes in Table 3. Recall that the Yeti controller needs to update *one and only one* (ingress) router when a session changes, which is independent of the topology size. The state for RB-OF and LIPSIN, on the other hand, grows with the topology size and number of receivers in the multicast sessions. Thus, as Table 3 shows, RB-OF and LIPSIN controllers need to update up to 103.3 and 19.6 core routers per each distribution graph change, respectively. That is, Yeti reduces the number of routers to be updated by up to 103X and 20X compared to RB-OF and LIPSIN, respectively.

To demonstrate the generality of Yeti performance, we calculate the average percentage of routers in the ISP topology to be updated for each change in the multicast distribution graph, which is taken over all 14 ISP topologies. We present

ISP Size	RB-OF	LIPSIN	Yeti	Saving (%)
49	18.9	4.9	1.0	94.7 / 79.7
84	55.9	12.3	1.0	98.2 / 92.0
125	76.8	19.6	1.0	98.7 / 95.0
158	91.2	11.8	1.0	98.9 / 91.5
197	103.3	18.7	1.0	99.0 / 94.7
Norm.	48 ± 10	9 ± 3	1 ± 0.6	97.5 / 87.1
Avg. (%)				

Table 3: Number of routers that need to be updated for each change in the multicast distribution graph. The shown savings in the right most column are relative to RB-OF and LIPSIN, respectively. The averages in the bottom row are computed across all 14 ISP topologies and normalized by the number of routers in each topology, and they represent the average percentage of routers to be updated for each change.

the results in the last row in Table 3, which show that Yeti reduces the average state update rate by 97.5% and 87.1% compared to RB-OF and LIPSIN, respectively.

In summary, compared to stateful and hybrid approaches, Yeti scales well and can handle dynamic multicast sessions, as it does not require maintaining state at any router, and it significantly reduces the need for frequent state updates.

5.3 Yeti vs A Stateless Approach

We compare Yeti against BIER-TE [3], which is a recent label-based multicast forwarding system. We implemented the basic features of BIER-TE as described in [3], which are the bit positions for the forward-connected, forward-routed and local-decap actions. This means that we *conservatively* report the minimum size of BIER-TE labels for every ISP topology. Since Yeti and BIER-TE are stateless and both use labels, we only analyze the label size and its imposed total overhead.

We first assess the label size per packet for multiple receiver densities. We present the CDF of the label size for the median and largest topologies in Figure 7; the results for other topologies are given in Appendix §E. Figure 7 indicates that the label size for Yeti is much smaller than that of BIER-TE in practical scenarios. For example, for the topology with the median number of routers (110 routers), Figure 7a, and receiver density of 30%, Yeti reduces the label size for 50% of the packets by 91.6% compared to BIER-TE. Moreover, the label size in Yeti for 90% of the packets is less than 19 bytes, while the label size of BIER-TE is 64 bytes. For the largest topology in our dataset (197 routers) and for receiver density of 30%, Figure 7b shows that the label size in Yeti is 15X smaller than BIER-TE for 50% of the packets.

We further analyze the behavior of Yeti and the dynamics of its label size as packets traverse the network. In Figure 8,

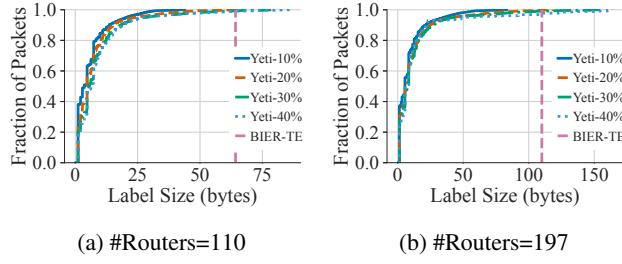


Figure 7: Label size CDF for different receiver densities.

we plot the average label size for Yeti and BIER-TE versus the number of hops from source, for the median and largest topologies, more results for other topologies are given in Appendix §E. The figure shows that the label size for BIER-TE depends on the topology size, it is 64 and 110 bytes, for the median and largest topologies, respectively. In contrast, the label size in Yeti decreases quickly as the packet moves away from the source. For example, in Figure 8a, the label size of Yeti is reduced by 16.9% and 67.8% after traversing 1 and 5 hops, respectively. The label size of Yeti becomes smaller than that of BIER-TE after traversing 2 hops only, and Yeti reduces the label size by 87.5% after traversing the first 50% of the hops.

Finally, we assess the total end-to-end label overhead of Yeti and BIER-TE, which we define as the label size multiplied by the number of network hops the packet traverses; this is the area under the curves in Figure 8. Table 4 summarizes the label overhead in bytes for multiple sample topologies. The results show that Yeti achieves substantial savings, up to 70.2%, compared to BIER-TE. In addition, we report the average label overhead per router across all 14 ISP topologies. On average, Yeti needs only 4 bytes per router to forward packets of multicast sessions, whereas BIER-TE requires 11.4 bytes per router, that is Yeti achieves an average saving of 65.3% in the label overhead.

In summary, the label size in Yeti quickly decreases as packets move towards the multicast destinations, because routers copy only a subset of labels for every branch. This results in substantial savings in the label overheads compared to the closest label-based multicast forwarding system, BIER-TE. In addition, BIER-TE cannot satisfy the service chaining requirements for multicast traffic, while Yeti can.

5.4 Analysis of Yeti

We analyze the performance of Yeti in terms of effectiveness of FSP labels, label size, processing overheads and running time to satisfy various forwarding requirements.

Effectiveness of FSP Labels. We study the importance of the proposed FSP label type. Recall that a single FSP label represents multiple routers in a path segment if that segment is on the shortest path. To show the importance of FSP label

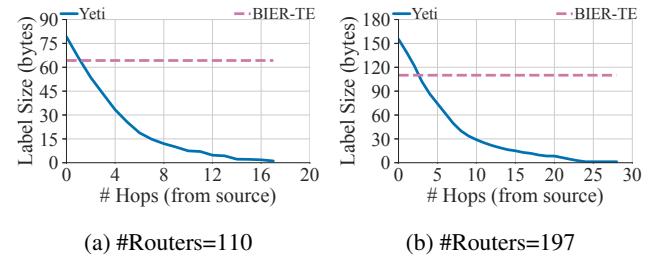


Figure 8: Label size as packets traverse the network.

ISP Size	BIER-TE	Yeti	Saving (%)
49	299.3	89.3	70.2
84	1,283.3	508.5	60.4
125	1,761.5	588.5	66.6
158	3,123.0	1,176.2	62.3
197	3,190.0	1,062.6	66.7
Norm. Avg. (bytes/router)	11.4 ± 4.8	4.0 ± 1.8	65.3

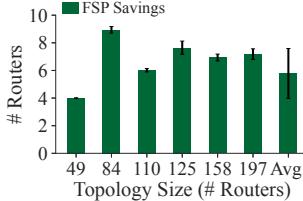
Table 4: Label overhead in bytes for Yeti and BIER-TE.

across different topologies, we plot the average FSP saving for sample topologies in Figure 9a as well as the average over all ISP topologies. We observe consistent savings across the topologies which range from 4 to 9 routers per FSP label. That is, one FSP label saves 4–9 other labels, reducing the label overhead by up to 9X. The average FSP saving is 5.8 ± 1.8 routers.

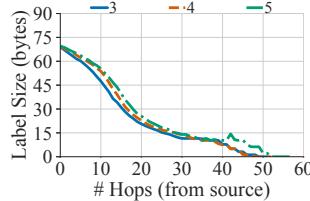
Next, we assess the FSP savings to satisfy service chaining requirements with different chain lengths in Table 5. As the results show, FSP labels efficiently encode path segments in the distribution graphs. For instance, an FSP label can encode about 6 routers on average for typical chain lengths.

Label Size. In Figure 9b, we plot the label size of Yeti versus the number of hops from source for service chaining requirements. The figure shows the results for the median topology of size 110. The results indicate that the label size of Yeti quickly decreases for all considered chain lengths as packets traverse towards the destinations. For instance, the label size decreases by 23% after traversing the first 10 hops when the chain length is 4. In addition, although the used routing policy [1] increases the number of hops to satisfy all service chaining requirements, Yeti is able to encode these large graphs in relatively small labels.

Processing Overhead. In Figure 10a, we plot the number of copy operations per packet versus the receiver density for multiple topology sizes to realize traffic engineering. The results for other ISP topologies are plotted in Appendix §E. The figure shows that the additional work per packet is small. The number of copy operations increases as the receiver density increases because the multicast distribution graphs have



(a) FSP savings



(b) Label size per hop

Figure 9: Analysis of FSP savings and label size of Yeti to satisfy traffic engineering and service chaining requirements.

ISP	Service Chain Length			
	Size	3	4	5
49	49	4.2±0.4	4.0±0.0	4.0±0.0
84	84	11.2±1.7	11.0±1.4	11.0±1.4
125	125	9.2±1.2	9.0±1.3	9.2±1.2
158	158	8.0±1.1	8.0±1.1	8.0±1.1
197	197	7.2±1.2	7.2±1.2	7.2±1.2
Avg.	Avg.	6.4±2.2	6.3±2.2	6.4±2.2

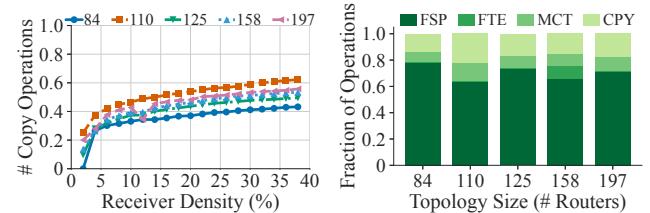
Table 5: Number of traversed routers per FSP label to satisfy service chaining.

more branches in this case. However, the processing overhead increases slowly. For instance, in the 84-router topology, the average number of copy operations increases from 0.4 to 0.62 per packet when the receiver density increases from 5% to 38%. This pattern applies for other topologies as well. That is, Yeti routers scale in terms of processing as the network load increases.

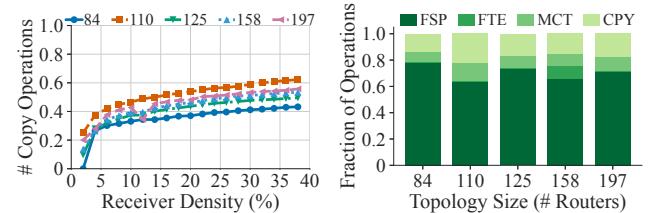
We next present the distribution of FSP, FTE, MCT and CPY labels per router for the five topologies in Figure 10b. The figure shows that the fraction of processed CPY labels (the most expensive) per Yeti router across all sessions is small compared to other label types. For example, only 17% of the labels being processed at a Yeti router are CPY labels for the largest topology of 197 routers.

For service chaining, Yeti incurs a similar distribution of operations to Figure 10b. For instance, the fraction of processed CPY labels is 18.7% for the topology of size 110 when the chain length is 3. For a longer chain of length 5, the fraction of CPY labels increases slightly to 21% to satisfy all service chains. On average, the fractions of CPY labels per ISP topology are $19.2\%\pm2.8\%$, $19.6\%\pm2.8\%$, and $19.6\%\pm2.3\%$ for chain lengths of 3, 4 and 5, respectively. We present the average number of copy operations and distribution of all operations in Appendix §E, where the averages are taken over chain lengths.

Running Time of Yeti Controller. We ran the proposed CREATELABELS algorithm on a workstation with four 3.3 GHz



(a) # copy operations



(b) Distribution of operations

Figure 10: Analysis of processing overheads of Yeti.

cores and 32 GB of memory, and we measured the running time of creating labels per graph update at the controller. The running time varied from 4 msec to 10 msec based on the topology size. For the largest topology of size 197, the controller spends only about 10 msec per graph update to create the labels for the largest session. Thus, the proposed label creation algorithm is practical, can run on commodity servers, and it supports frequent graph updates and network dynamics.

In summary, Yeti imposes small label and processing overheads while satisfying service chaining and traffic engineering requirements.

6 Conclusions

We proposed an efficient, stateless, multicast forwarding system called Yeti that implements generalized multicast graphs in ISP networks. Unlike current rule-based multicast systems, Yeti does not require maintaining any state at routers. And unlike other label-based multicast systems, Yeti can direct traffic on arbitrary network paths to meet traffic engineering and service chaining requirements while reducing the label size significantly. The novel aspects of Yeti include (1) supporting general traffic forwarding requirements, (2) guaranteeing correctness, (3) composing small labels, and (4) processing labels efficiently at routers. We implemented Yeti in a programmable router and evaluated its performance. Our experiments show that Yeti can achieve line-rate performance while using a small amount of hardware resources. In addition, we conducted extensive simulations using real ISP topologies, and compared it versus the state-of-art approaches. Our results show that Yeti outperforms the other approaches by wide margins for all considered metrics.

Acknowledgments

We thank our shepherd, Behnaz Arzani, and the anonymous reviewers for their comments. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

- [1] K. Diab, C. Lee, and M. Hefeeda. Oktopus: Service chaining for multicast traffic. In *Proc. of IEEE ICNP’20*, pages 1–11, Madrid, Spain, October 2020.
- [2] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. The lockdown effect: Implications of the covid-19 pandemic on internet traffic. In *Proc. of ACM IMC’20*, page 1–18, Virtual Event, October 2020.
- [3] Toerless Eckert, Gregory Cauchie, and Michael Menth. Tree Engineering for Bit Index Explicit Replication (BIER-TE). Internet-Draft draft-ietf-bier-te-arch-08, Internet Engineering Task Force, July 2020. Work in Progress.
- [4] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. Segment Routing Architecture. RFC 8402.
- [5] B. Ren, D. Guo, G. Tang, X. Lin, and Y. Qin. Optimal service function tree embedding for nfv enabled multicast. In *Proc. of IEEE ICDCS’18*, pages 132–142, Vienna, Austria, July 2018.
- [6] Jiaqi Zheng, Bo Li, Chen Tian, Klaus-Tycho Foerster, Stefan Schmid, Guihai Chen, and Jie Wu. Scheduling congestion-free updates of multiple flows with chronicle in timed sdns. In *Proc. of IEEE ICDCS’18*, pages 12–21, Vienna, Austria, July 2018.
- [7] S. H. Chiang, J. J. Kuo, S. H. Shen, D. N. Yang, and W. T. Chen. Online multicast traffic engineering for software-defined networks. In *Proc. of IEEE INFOCOM’18*, pages 414–422, Honolulu, HI, April 2018.
- [8] Aravindh Raman, Gareth Tyson, and Nishanth Sastry. Facebook (A)Live?: Are Live Social Broadcasts Really Broadcasts? In *Proc. of WWW’18*, page 1491–1500, Lyon, France, April 2018.
- [9] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K. Reiter, and Vyas Sekar. Intent-driven composition of resource-management sdn applications. In *Proc. of ACM CoNEXT’18*, pages 86–97, Heraklion, Greece, 2018.
- [10] IJsbrand Wijnands, Eric C. Rosen, Andrew Dolganow, Tony Przygienda, and Sam Aldrin. Multicast Using Bit Index Explicit Replication (BIER). RFC 8279.
- [11] L. H. Huang, H. C. Hsu, S. H. Shen, D. N. Yang, and W. T. Chen. Multicast traffic engineering for software-defined networks. In *Proc. of IEEE INFOCOM’16*, pages 1–9, San Francisco, CA, April 2016.
- [12] M. Huang, W. Liang, Z. Xu, W. Xu, S. Guo, and Y. Xu. Dynamic routing for network throughput maximization in software-defined networks. In *Proc. of IEEE INFOCOM’16*, pages 1–9, San Francisco, CA, April 2016.
- [13] Bill Fenner, Mark J. Handley, Hugh Holbrook, Isidor Kouvelas, Rishabh Parekh, Zhaohui (Jeffrey) Zhang, and Lianshu Zheng. Protocol Independent Multicast - Sparse Mode: Protocol Specification. RFC 7761.
- [14] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proc. of ACM IMC’15*, pages 275–287, Tokyo, Japan, October 2015.
- [15] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.
- [16] S. H. Shen, L. H. Huang, D. N. Yang, and W. T. Chen. Reliable multicast routing for software-defined networks. In *Proc. of IEEE INFOCOM’15*, pages 181–189, Hong Kong, China, April 2015.
- [17] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, Jan 2015.
- [18] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *Proc. of ACM SIGCOMM’15*, pages 15–28, London, United Kingdom, 2015.
- [19] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, 34(5):32–41, September 2014.
- [20] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *Proc. of ACM SIGCOMM’14*, pages 539–550, Chicago, IL, August 2014.
- [21] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.
- [22] X. Chen, M. Chen, B. Li, Y. Zhao, Y. Wu, and J. Li. Celery: A low-delay multi-party conferencing solution. *IEEE Journal on Selected Areas in Communications*, 31(9):155–164, September 2013.

- [23] Bob Thomas, IJsbrand Wijnands, Ina Minei, and Kireeti Kompella. LDP Extensions for Point-to-Multipoint and Multipoint-to-Multipoint Label Switched Paths. RFC 6388.
- [24] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *Proc. of ACM Workshop on Networking Meets Databases (NetDB'11)*, Athens, Greece, June 2011.
- [25] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. Lipsin: Line speed publish/subscribe inter-networking. In *Proc. of ACM SIGCOMM'09*, pages 195–206, Barcelona, Spain, August 2009.
- [26] T. W. Cho, M. Rabinovich, K. K. Ramakrishnan, D. Srivastava, and Y. Zhang. Enabling content dissemination using efficient and scalable multicast. In *Proc. of IEEE INFOCOM'09*, pages 1980–1988, Rio de Janeiro, Brazil, April 2009.
- [27] V. Gopalakrishnan, B. Bhattacharjee, K. K. Ramakrishnan, R. Jana, and D. Srivastava. Cpm: Adaptive video-on-demand with cooperative peer assists and multicast. In *Proc. of IEEE INFOCOM'09*, pages 91–99, Rio de Janeiro, Brazil, April 2009.
- [28] Bradley Cain, Dr. Steve E. Deering, Bill Fenner, Isidor Kouvelas, and Ajit Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376.
- [29] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the ip multicast service and architecture. *IEEE Network*, 14(1):78–88, January 2000.
- [30] Wen-Tsuen Chen, Pi-Rong Sheu, and Yaw-Ren Chang. Efficient multicast source routing scheme. *Computer Communications*, 16(10):662–666, 1993.
- [31] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.
- [32] AWS Announces Nine New Compute and Networking Innovations for Amazon EC2. <https://bloom.bg/2t1N9py>. [Online; accessed February 2022].
- [33] Run IP Multicast Workloads in the Cloud Using AWS Transit Gateway. <https://go.aws/2RT6stz>. [Online; accessed February 2022].
- [34] NetFPGA SUME Reference Learning Switch Lite. <https://bit.ly/2UrUFlx>. [Online; accessed February 2022].
- [35] The Internet Topology Zoo. <http://www.topology-zoo.org/dataset.html>. [Online; accessed February 2022].
- [36] Apache ActiveMQ. <http://activemq.apache.org>. [Online; accessed February 2022].
- [37] Bt iptv (youview). <https://bit.ly/3ssCvTz>. [Online; accessed February 2022].
- [38] Multicast Command Reference for Cisco ASR 9000 Series Routers. <https://bit.ly/3AaVGDQ>. [Online; accessed February 2022].
- [39] RabbitMQ. <http://www.rabbitmq.com>. [Online; accessed February 2022].
- [40] U-verse tv. <https://bit.ly/3HLhfyl>. [Online; accessed February 2022].
- [41] Zuckerberg really wants you to stream live video on Facebook. <https://bit.ly/2v6uHqF>. [Online; accessed February 2022].
- [42] Benoit Donnet, Korian Edeline, Iain R. Learmonth, and Andra Lutu. Middlebox classification and initial model. <https://bit.ly/3dZelXV>. [Online; accessed February 2022].

Appendix A Correctness of Yeti

Theorem 1 (Correctness). *Yeti forwards packets on and only on links that belong to the multicast graph.*

Proof. Yeti guarantees correctness by creating an ordered set of Yeti labels for the given graph at the controller using the ENCODEGRAPH algorithm. Recall that the algorithm first creates a tree to represent the services needed before reaching the destinations. A node in that calculated tree consists of router ID v and sub-sequence of services \mathbb{S} . Every node appears only once in the tree (by construction). This means that the tree has no cycles.

The label creation algorithm traverses the tree to calculate the final labels. Within every path with provided services \mathbb{S} , the order and type of created labels represent how packets should be forwarded in the data plane. This is detailed as follows. FSP labels do not result in incorrect forwarding because: (1) an FSP label with ID v is only added when a tree node v is traversed by the CREATELABELS algorithm, (2) since every node with router ID v and services \mathbb{S} is traversed once and only once by the algorithm, only a single FSP v can be added to the labels representing that node with services \mathbb{S} , (3) in the data plane, the router with ID v removes the FSP v label. Thus, no subsequent routers in along the path with same services can process that label and transmit the packet back to v , and (4) since the traversal starts from the source, if

a node u precedes node v in the tree, the algorithm guarantees that u is traversed before v . Thus, there is no label to forward packets back to u . Similar properties are guaranteed for FTE labels in terms of links. Moreover, attaching multiple FTE labels does not result in incorrect forwarding. Otherwise, the given tree has loops, or routers do not remove FTE labels.

For MCT and CPY labels, since the CREATELABELS algorithm recursively creates the labels for each branch, the same guarantees apply within a single branch for branching points. In addition, routers in one branch do not forward packets to routers in other branches. This is because (1) the given tree is a proper one (i.e., has no cycles), and (2) every router in a given branch processes the subset of labels duplicated for that branch using the CPY labels. \square

Appendix B Practical Considerations of Yeti

Multicast across ISPs. The description of Yeti thus far has focused on offering a scalable multicast service within a *single* ISP. Extending Yeti to multiple ISPs can be done in multiple ways. For example, a content provider could have separate agreements with different ISPs to serve clients within these ISPs, where each ISP runs its multicast service independently from the others. In this case, a separate feed of the multicast session traffic is provided from the content provider to each ISP. Agreements between major content providers, e.g., Facebook and Netflix, and large ISPs are not uncommon. Another way of extending Yeti to multiple ISPs is through tunneling, where a tunnel is established between an egress router of an ISP to an ingress router of another ISP. The ingress router of the second ISP would attach labels created by the controller of that ISP. While the tunneling approach does not reveal the internal network details of ISPs to each other, which is important in practice, it does require collaboration among ISPs to establish tunnels among some routers.

Incremental Deployment. An ISP may have some legacy routers that are not programmable and thus cannot run the packet processing algorithm of Yeti. There are multiple options that Yeti can still function in this situation, albeit with some workaround and minor overheads. First, Yeti is general and can support arbitrary multicast graphs. Thus, a possible solution is to modify the multicast graphs to avoid going through legacy routers. The multicast graphs is an *input* to our label creation algorithm, and thus the computed labels will not direct traffic through legacy routers. If a legacy router cannot be avoided, a tunnel can be created between the router immediately before the legacy router and each router following it has multicast destinations.

Appendix C Illustrative Example

We present a simple example to illustrate all steps of the proposed approach. Figure 11 shows the multicast tree of the

session in Figure 1, where solid arrows indicate the graph links. The dotted line is the shortest path that the ISP avoids because it is over-utilized. Router IDs and used interface IDs are shown in the figure. The number of core routers and maximum interface count are 12 and 5, respectively. Thus, the label sizes (in bits) are 8, 8, 7 and 5 for MCT, CPY, FSP and FTE, respectively (Table 1).

The controller generates the shown labels using the ENCODEGRAPH algorithm as follows. First, the algorithm creates three FSP labels to encode path segments to routers 2, 7 and 4. Notice that the most significant bit in each of these labels is set to one as these nodes provide services a, b and c.

The algorithm then generates MCT 1-00110 to duplicate packets on interfaces 2 and 3 at router 4. Since the children 3 and 7 have core children, the algorithm sets the most significant bit in the MCT label to one, and creates two CPY labels for branches A and B. In branch B, the recursive call of Algorithm 1 creates labels for the path segment {3, 6, 5, 8} as follows. First, the algorithm appends routers 3, 6 and 5 to *pth_seg* because each has one core child (Line 13, Algorithm 1). When the algorithm reaches 8 (which has two core children), the algorithm appends it to *pth_seg* (Line 25, Algorithm 1) and creates labels for the path segment and the branching point at router 8. For the path segment, since link (3, 6) is not on the shortest path from 3 to 8, the algorithm creates FTE 011 to forward packet on interface 3 at router 3. In addition, the algorithm creates FSP 0-1000 to forward packets from 6 to 8, because the links (6, 5) and (5, 8) are on the shortest path between 3 and 8.

We describe the packet processing algorithm at representative routers. The dark labels in Figure 11 are the ones that are processed at given routers. When router 2 receives FSP 1-0010, it decides that the packet needs to be processed by service a. So, it removes the label and forwards the packet to the corresponding datacenter. When the processing is done, router 2 receives a packet which its first label is FSP 1-0111. Thus, it forwards the packet on the shortest path to router 7. Notice that router 4 does not remove FSP 1-0111 as it is not destined for it. Then, routers 7 and 4 receives packets where the contents of the FSP labels are their router IDs. After service c processes the packet, router 4 processes MCT 1-00110 by duplicating the packet on interfaces 2 and 3, and copying specific byte ranges using the CPY labels. Router 3 forwards the packet on interface 3. Router 8 removes FSP and MCT labels and duplicates the packet to routers 11 and 12. Since the packet has no labels at router 11 and 12, they transmit it to egress routers.

Appendix D Implementation of Yeti using P4

P4 is a data plane programming language that is getting popular due to its flexibility. Thus, we show that Yeti can be implemented in P4 switches. We implemented Yeti using the Intel P4 software development environment (SDE) version 9.5.0.

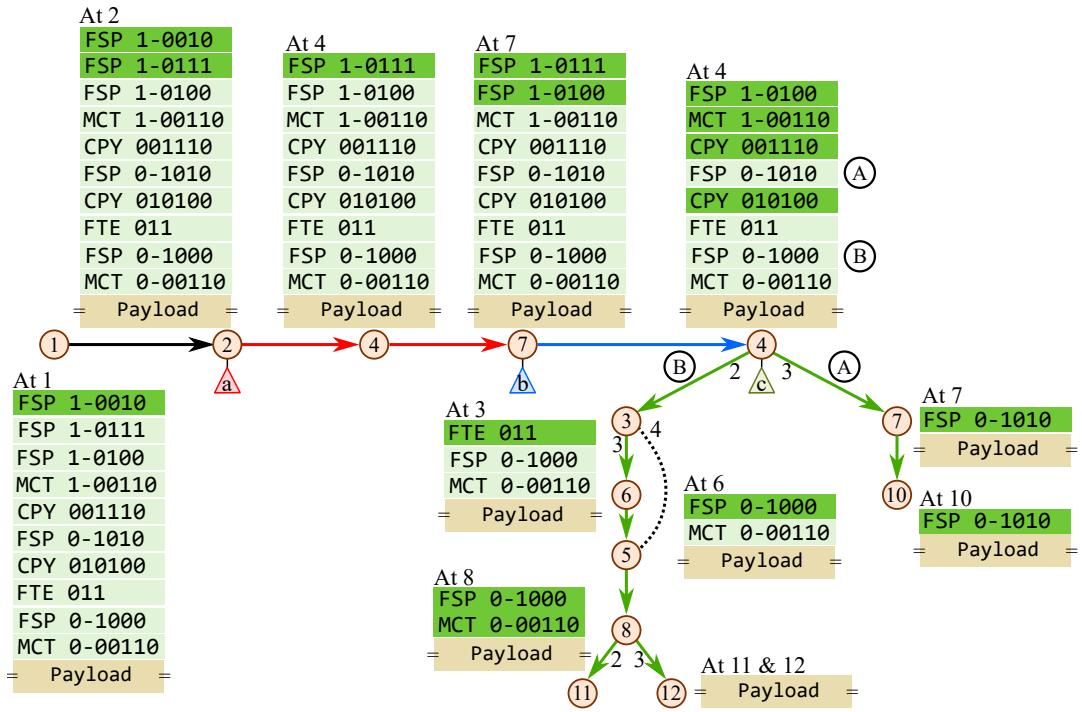


Figure 11: Illustrative example of how labels in Yeti represent the multicast tree in Figure 1.

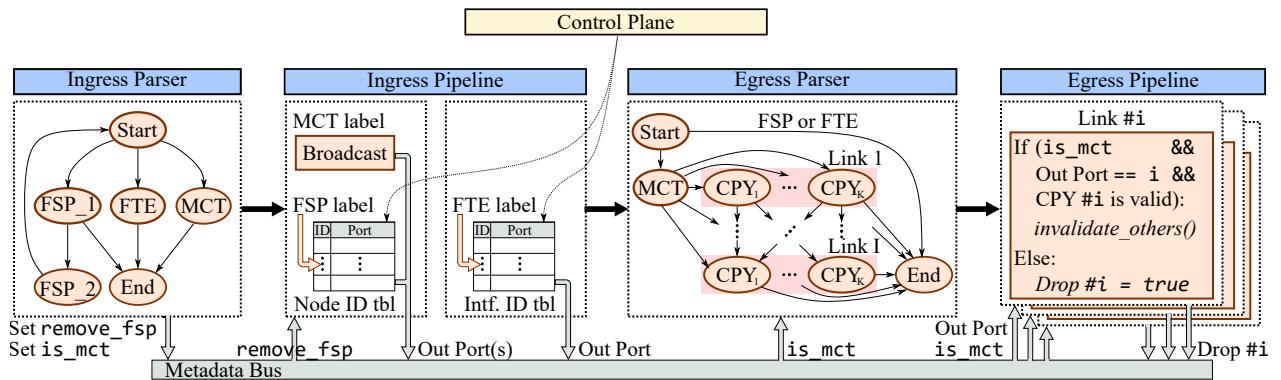


Figure 12: Design of Yeti in P4 switches.

We used the provided tools such as the P4 compiler (called `bf-p4c`) and switch model integrated with the SDE to realize our ideas and validate our implementation. We note that P4 programs implemented in open-source software switches, e.g., `bmv2`, may not necessarily work on actual hardware switches, because of the potential mismatch between the available physical resources in actual switches and the assumed resources in software switches. This is not the case for the Intel SDE, since its P4 compiler produces code for actual Tofino switches (which are also manufactured by Intel).

There are two main challenges in implementing Yeti using P4. First, current Tofino switches do not provide programmable primitives to implement new multicast systems. This is because the packet replication engine is implemented as a fixed-function block. Second, the current `bf-p4c` compiler does not support variable-length fields (i.e., `varbit`) needed to parse and process CPY labels. We made three design choices to address these challenges. We first divided the packet processing between ingress and egress pipelines to realize a programmable multicast primitive. Second, we relied on the available resources and programmable capabilities of the parsers to reduce the processing in ingress/egress stages. Finally, we explicitly unrolled the parsing and processing of a CPY label as an array of items.

Figure 12 illustrates the high-level design of our P4 implementation. Upon a packet arrival, the ingress parser processes FSP, FTE and MCT labels as follows. For an FSP label, the parser reads the included node ID, and parses the labels again if the parsed node ID is the same as the router ID. In this case, the parser sets a metadata field `remove_fsp` to be used by the ingress pipeline. Recall that the Yeti controller never creates two consecutive FSP labels (§3.4 and §3.6). Therefore, the parser always terminates. In the case of FTE and MCT labels, the parser reads their contents. In addition, it sets a metadata field `is_mct` when it parses an MCT label.

The ingress pipeline contains two tables to maintain node and interface IDs, and spans two stages. In the first stage, the algorithm processes an FSP label by reading an entry from the node ID table that matches the label content, and setting the outgoing port accordingly. The algorithm also removes the FSP label if the metadata field `remove_fsp` is set. The algorithm processes MCT labels, in the same stage,

by duplicating the packet to all outgoing ports. The FTE label processing is done in the second stage, and it is similar to that of FSP. However, the algorithm always removes FTE labels.

The egress parser and pipeline are used only to handle MCT and CPY labels. For each duplicated packet, the egress parser extracts the contents of MCT label when the metadata field `is_mct` is set. The parser then reads CPY labels sequentially based on the content of MCT label and offsets in CPY labels. Specifically, for each CPY label, the parser extracts its offset and content. The content is read based on the offset value, and represented as an array of up to K multiples of B bits to emulate `varbit<K×B>`. When the parser is done, the egress pipeline identifies which outgoing port should transmit what CPY label based on the egress port number and validity of the CPY label. The MCT and remaining CPY labels are removed.

We did not have a physical Tofino switch in our lab to conduct measurement experiments at the time of conducting this research. We thus validated our implementation by writing and running multiple test cases, and sending and receiving packets to and from the Tofino switch model process. When we run a test case, we insert the required entries into node and link IDs tables. We send labeled packets using `scapy`, and verify the reception of outgoing packets based on the attached Yeti labels. A test case succeeds if all packets were received on and only on expected ports with expected headers.

Appendix E Additional Simulation Results

This appendix includes more figures and results from our simulation.

Figure 13 shows the state size for all 12 ISP topologies. The figure indicates that Yeti provides a scalable multicast service as it does not require any state at any router.

Figures 14–15 show the label size of Yeti versus BIER-TE. Compared to BIER-TE, the figures show that Yeti reduces the label size significantly across receiver densities and as packets traverse the network.

Figures 16–19 indicate that Yeti impose small label and processing overheads when supporting service chaining and traffic engineering requirements.

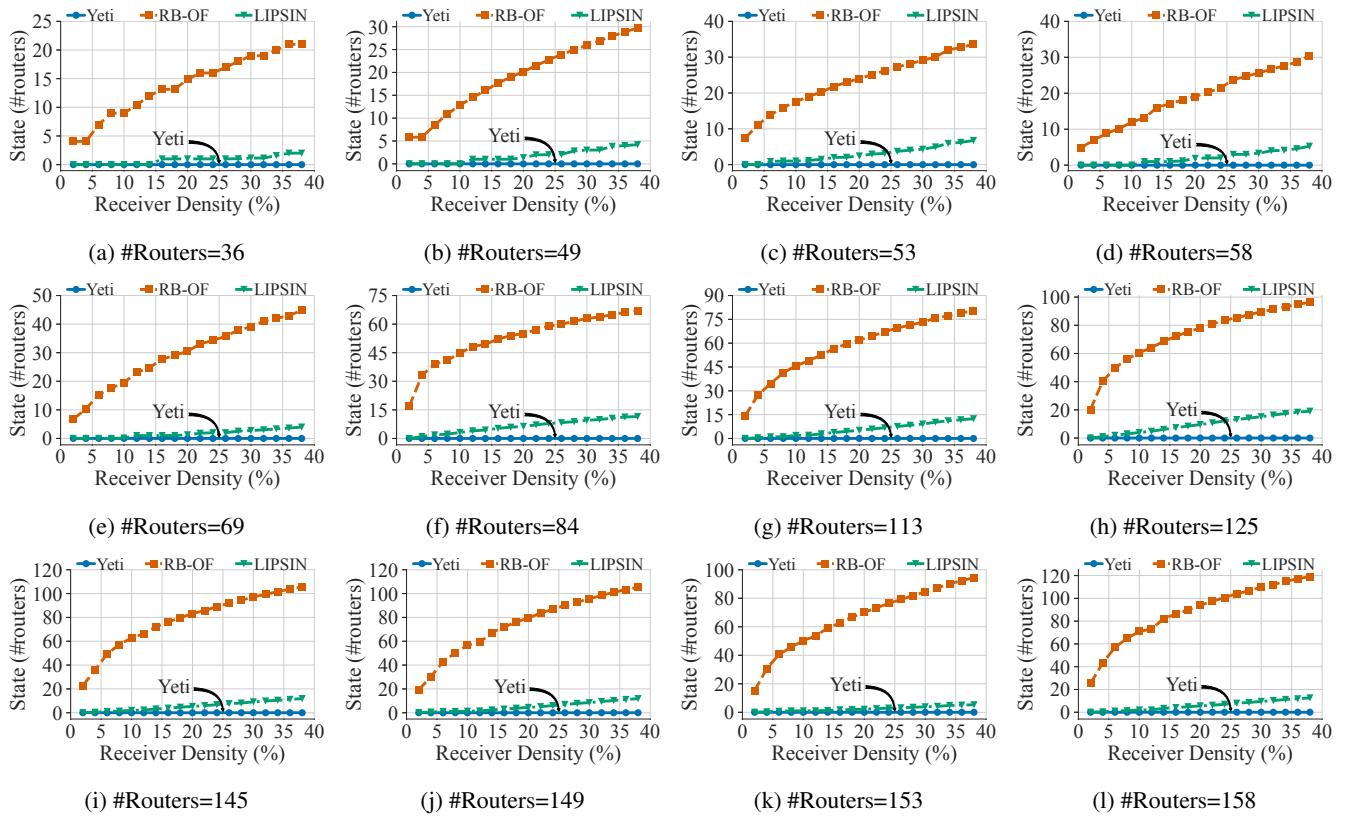


Figure 13: State size for the considered ISP topologies.

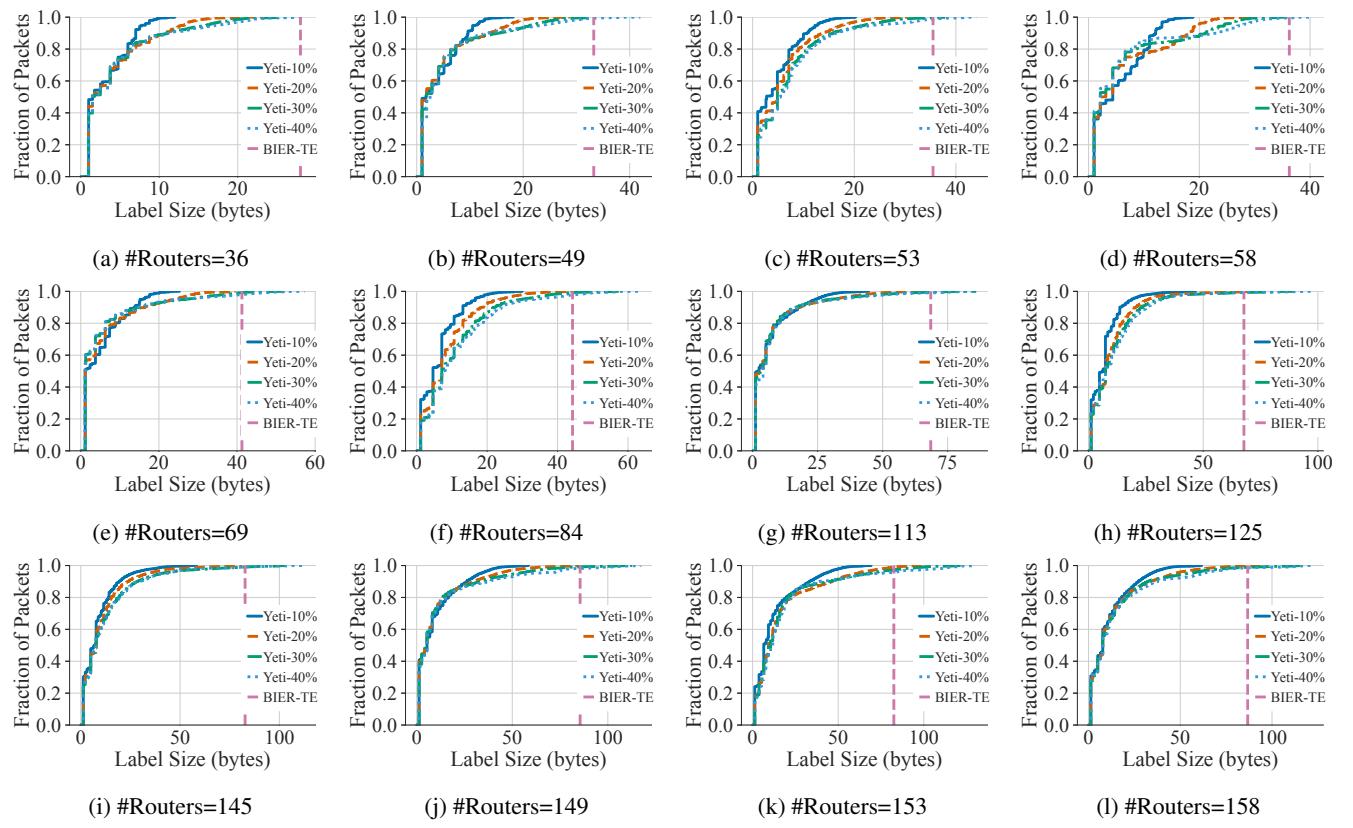


Figure 14: Label size CDF for different ISP topologies.

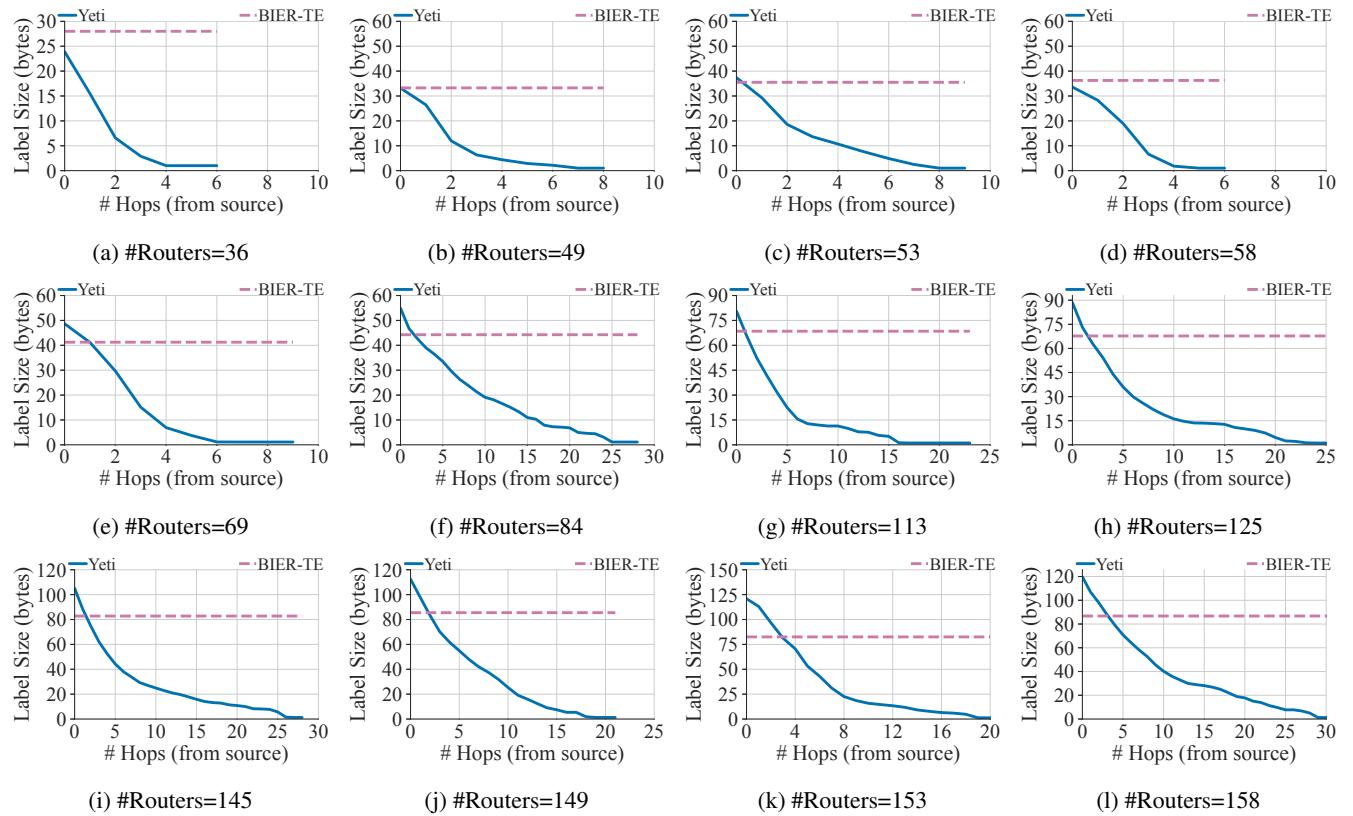


Figure 15: Label size of Yeti versus BIER-TE as packets traverse the network.

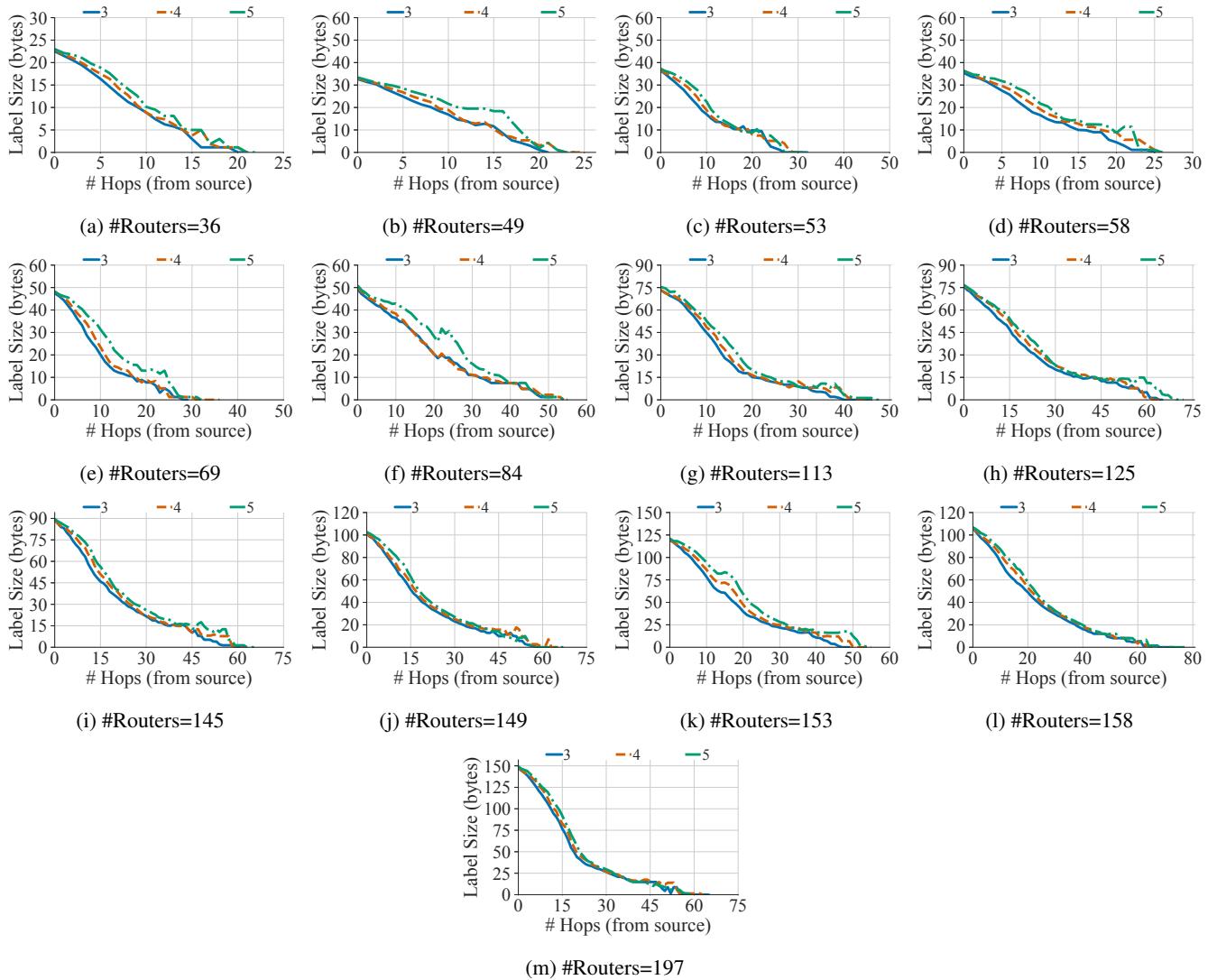


Figure 16: Analysis of label size of Yeti to satisfy service chaining requirements.

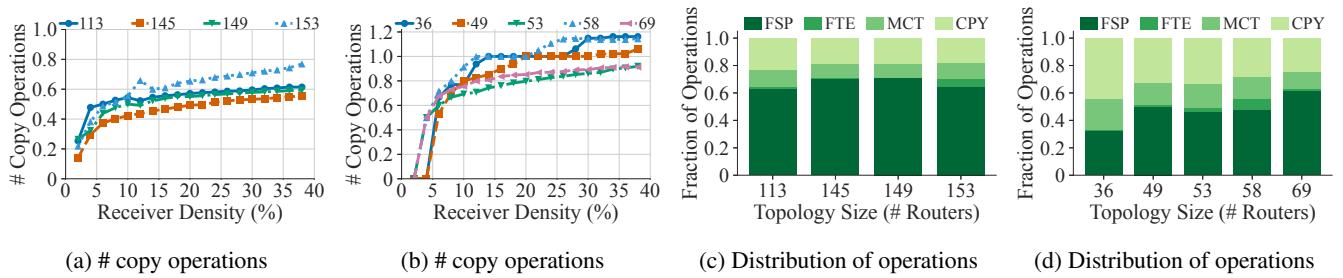


Figure 17: Analysis of processing overheads of Yeti for different ISP topologies.

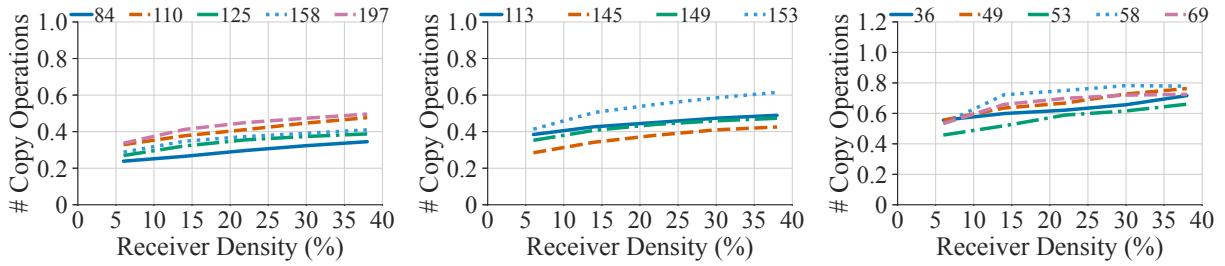


Figure 18: # copy operations for different ISP topologies to support service chaining.

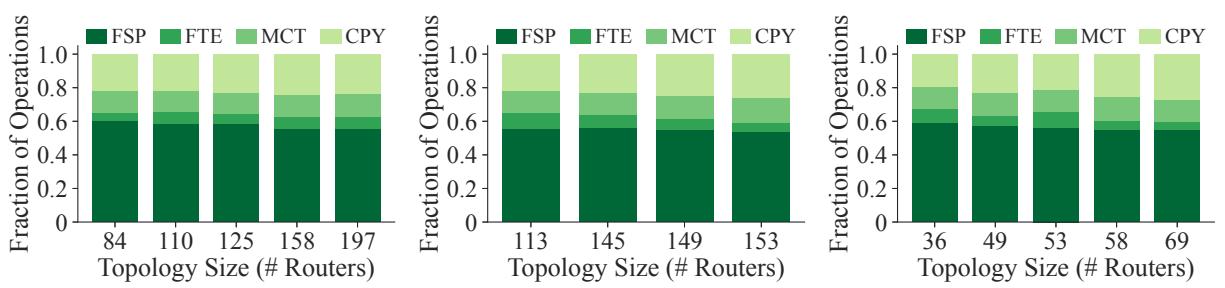


Figure 19: Distribution of FSP, FTE, MCT and CPY operations to support service chaining.

cISP: A Speed-of-Light Internet Service Provider

Debopam Bhattacherjee*¹, Waqar Aqeel*², Sangeetha Abdu Jyothi^{3,4}, Ilker Nadi Bozkurt^{2†}, William Sentosa⁵, Muhammad Tirmazi⁶, Anthony Aguirre⁷, Balakrishnan Chandrasekaran⁸, P. Brighten Godfrey^{5,9}, Gregory Laughlin¹⁰, Bruce Maggs^{2,11}, Ankit Singla¹

¹ETH Zürich, ²Duke University, ³UC Irvine, ⁴VMware Research, ⁵UIUC, ⁶Harvard University, ⁷UC Santa Cruz, ⁸VU Amsterdam, ⁹VMware,
¹⁰Yale University, ¹¹Emerald Technologies

Abstract

Low latency is a requirement for a variety of interactive network applications. The Internet, however, is not optimized for latency. We thus explore the design of wide-area networks that move data at nearly the speed of light in vacuum. Our **cISP** design augments the Internet’s fiber with free-space microwave wireless connectivity over paths very close to great-circle paths. cISP addresses the fundamental challenge of simultaneously providing ultra-low latency while accounting for numerous practical factors ranging from transmission tower availability to packet queuing. We show that instantiations of cISP across the United States and Europe would achieve mean latencies within 5% of that achievable using great-circle paths at the speed of light, over medium and long distances. Further, using experiments conducted on a nearly-speed-of-light algorithmic trading network, together with an analysis of trading data at its end points, we show that microwave networks are *reliably* faster than fiber networks even in inclement weather. Finally, we estimate that the economic value of such networks would substantially exceed their expense.

1 INTRODUCTION

User experience in many interactive network applications depends crucially on achieving low latency. Even seemingly small increases in latency can negatively impact user experience, and, subsequently, revenue for service providers: Google, for example, quantified the impact of an additional 400 ms of latency in search results as 0.7% fewer searches per user [18]. Further, wide-area latency is often the bottleneck, as Facebook’s analysis of over a million requests found [21]. Indeed, content delivery networks (CDNs) present latency reduction and its associated increase in conversion rates as one of the key value propositions of their services, citing, e.g., a 1% loss in sales per 100 ms of latency for Amazon [2]. In spite of the significant impact of latency on performance and user experience, the Internet is not designed to treat low latency as a primary objective. This is the problem we address: reducing latencies over the Internet to the lowest possible.

The best achievable latency between two points along the surface of the Earth is determined by their geodesic distance divided by the speed of light, c . Latencies over the Internet, however, are usually much larger than this minimal “ c -latency”: recent measurement work found that fetching even small amounts of data over the Internet typically takes $37\times$ longer than the c -latency, and often, more than $100\times$ longer [16]. This delay comes from the many round-trips between the communicating endpoints, due to inefficiencies in the transport and application layer protocols, and from each round-trip itself taking $3-4\times$ longer than the c -latency [16]. Given the approximately multiplicative role of network round-trip times (RTTs) when bandwidth is not the main bottleneck, eliminating inflation in Internet RTTs can potentially translate to up to $3-4\times$ speedup, even *without* any protocol changes. Further, as protocol stack improvements get closer to their ideal efficiency of one RTT for small amounts of data, the RTT becomes the singular network bottleneck. Similarly, for well-designed applications dependent on persistent connectivity between two fixed locations, such as gaming, *nothing* other than resolving this $3-4\times$ “infrastructural inefficiency” can improve latency substantially.

Thus, beyond the networking research community’s focus on protocol efficiency, reducing the Internet infrastructure’s latency inflation is the next frontier in research on latency. While academic research has typically treated infrastructural latency inflation as an unresolvable given, we argue that this is a high-value opportunity, and is much more tractable than may be evident at first.

What are the root causes of the Internet’s infrastructural inefficiency, and how do we ameliorate them? Large latencies are partly explained by poor use of existing fiber infrastructure: two communicating sites often use a longer, indirect route because their service providers do not peer over the shortest fiber connectivity between their locations. We find, nevertheless, that even latency-optimal use of *all* known fiber conduits, computed via shortest paths in the InterTubes dataset [34], would leave us $1.98\times$ away from c -latency [17]. This gap stems from the speed of light in fiber being $\sim\frac{2}{3}c$,

* Equal contribution. † Now at Google.

and the unavoidable circuitousness of fiber routes due to topographic and economic constraints of buried conduits.

We thus explore the design of **cISP**, an Internet Service Provider that provides nearly speed-of-light latency by exploiting wireless electromagnetic transmissions, which can be realized with point-to-point microwave antennas mounted on towers. This approach holds promise for overcoming both the aforementioned shortcomings fundamental to today’s fiber-based networks: the transmission speed in air is essentially equal to c , and the richness of existing tower infrastructure makes more direct paths possible. Nevertheless, it also presents several new challenges, including:

- overcoming numerous practical constraints, such as tower availability, line-of-sight requirements, and the impact of weather on performance;
- coping with limited wireless bandwidth;
- solving a large-scale cost-optimal network design problem, which is NP-hard; and
- addressing switching and queuing delays, which are more prominent with the smaller propagation delays.

To meet these challenges, we propose a hybrid design that augments the Internet’s fiber connectivity with nearly straight-line wireless links. These low-latency links are used judiciously where they provide the maximum latency benefit, and only for the high-impact but small proportion, in terms of bytes, of Internet traffic that is latency-sensitive. We design a simple heuristic that achieves near-optimal results for the network design problem. Our approach is flexible and enables network design for a variety of deployment scenarios; in particular, we show that cISP’s design for interconnecting large population centers in the contiguous U.S. and Europe can achieve mean latencies as low as $1.05 \times c$ -latency at a cost of under \$1 per gigabyte (GB). We show through simulation that such networks can be operated at high utilization without excessive queuing.

To address the practical concerns, we use fine-grained geographic data and the relevant physical constraints to determine where the needed wireless connectivity would be feasible to deploy, and assess our design under a variety of scenarios with respect to budget, tower height and availability, antenna range, and traffic matrices. We also use a year’s worth of meteorological data to assess the network’s performance during weather disturbances, showing that most of cISP’s latency benefits remain intact throughout the year. Our weather simulation and an animation showing how the hybrid network evolves from mostly-fiber to mostly-wireless with increasing budget are available online; see [25] and [26].

But is it feasible to use microwave hardware for low latency in practice? To answer this question, we rented virtual machines in the CME data center in Chicago and the Equinix data center in New Jersey, and, on Saturdays, were given access at these data centers to one of the fastest microwave networks spanning the Chicago – New Jersey algorithmic trading corri-

dor. Experiments conducted on this network show that it successfully operates at a speed extremely close to the speed of light, and that losses can be effectively handled by extremely lightweight forward error correction (FEC). We complement these findings by analyzing real trading data, revealing the minimum latency between the data centers and showing that the network is available in varied weather conditions.

Finally, we explore the application-level benefits for Web browsing and gaming, and present estimates showing that the utility of cISP vastly exceeds its cost, even for web sites already using CDNs to reduce latency.

2 TECHNOLOGY BACKGROUND

At the highest level, our approach involves using free-space communication between transmitters mounted at a suitable height, e.g., using dedicated towers or existing buildings, and separated from each other by at most a certain limiting distance. Network links longer than this range require a series of such transmitters. Typically, even after accounting for terrain, such a network link can be built close to the shortest path on the Earth’s surface between the two end points. Further, the speed of light in air is essentially the same as that in vacuum, c . These properties make our approach attractive for the design of (nearly) c -latency networks.

Technology choices. Several physical layer technologies are amenable for use in our design, including free-space optics (FSO), microwave (MW), and millimeter wave (MMW). At present, we believe MW provides the best combination of range, resilience, throughput, and cost. Future advances in any of these technologies, however, can be easily rolled into our design, and can only improve our cost-benefit analysis.

While hollow fiber [31] could, in the future, also provide c -latency, it would still suffer from the circuitousness of today’s fiber conduits. Low Earth orbit satellite networks, as are being currently deployed, could also help, although they currently incur substantially higher latency than cISP (§9).

Switching latency. While long-haul MW networks have existed since the 1940s [10], their use in high-frequency trading starting within the last 10 years [55] has driven innovation in radios so that each MW retransmission only takes a few μs . Thus, even wide-area links with many retransmissions incur negligible switching latency. As an example, the HFT industry operates a MW relay between Chicago and New Jersey comprising ≈ 20 line-of-sight links that operates within 1% of c -latency end-to-end at the *application* layer [58].

Packet loss. Loss occurs for several reasons, including weather disruption and intermittent multi-path fading, especially over bodies of water. In §5.1, using a year’s worth of weather data, we analyze the impact of diverting traffic to alternate (fiber or MW) routes during inclement weather. Our active experiments on a microwave network also show that losses experienced could be handled with lightweight forward error correction (FEC).

Spectrum and licensing. We propose the use of MW com-

munication in the 6-18 GHz frequency range. These frequencies are not very crowded, and licensing is generally not very competitive, except at 6 GHz in cities, and along certain routes, like the above mentioned HFT corridor. The licenses are given on a first-come, first-served basis, recorded in a public database, they protect against the deployment of other links that would interfere with licensed links.

Line-of-sight & range. Successive MW towers need line-of-sight visibility, accounting for the Earth’s curvature, terrain, trees, buildings and other obstructions, and atmospheric refraction. Attenuation also limits range. A maximum range of around 100 km is practicable, but we show results with maximum allowed range varying between 60-100 km (§5.2).

Bandwidth. Between any two towers, using very efficient encoding (256 QAM or higher), wide frequency channels, and radio multiplexing, a data rate of about 1 Gbps is achievable [45]. This bandwidth is vastly smaller than for fiber, and necessitates a hybrid design using fiber and MW.

Geographic coverage. Connecting individual homes directly to such a MW network would be cost-prohibitive. To maximize cost-efficiency, we focus on long-haul connectivity, with the last mile being traditional fiber. At short distances, fiber’s circuitousness and refraction are small overheads.

Cost model. We rely on cost estimates in recent work [55] and based on our conversations with industry participants involved in equipment manufacturing and link provisioning. The cost of installing a bidirectional MW link, on existing towers, is approximately \$75K (\$150K) for 500 Mbps (1 Gbps) bandwidth. The average cost for building a new tower is \$100K, with wide variation by terrain and across cities and rural areas. Any additional towers needed to augment bandwidth for particular links incur this “new tower” cost. The operational costs comprise several elements, including management and personnel, but the dominant operational expense, by far, is tower rent: \$25 – 50K per year per tower. We estimate cost per GB by amortizing the sum of building costs and operational costs over 5 years.

Note that the deployment and operational costs can vary substantially based on the deployment model. For example, imagine that a company like American Tower [7], which has a substantial tower presence across the US (see Fig. 14 in Appendix D), deploys cISP. In such a scenario, not only would the cost of bandwidth augmentation be negligible, but also the cost of maintaining the towers would be drastically reduced. We consider both conservative and optimistic deployment models and conduct an in-depth cost-analysis in this work.

3 CISP DESIGN

At an abstract level, given the tower and fiber infrastructure, a set of n sites (e.g., cities, data centers) to interconnect, and a traffic model between them, we want to select a set of tower-level connections that minimizes network-wide latency while adhering to a budget and the constraints outlined in §2. Our approach comprises the following three broad steps.

1. Identifying a set of links that are likely to be useful by determining, for each pair of sites (s, d) , the best feasible tower-level connectivity, if s and d were to be directly connected by a series of towers.
2. Building all $O(n^2)$ direct links, connecting each site to every other, would be prohibitively expensive. Thus, a subset of site-to-site links, together with existing fiber conduits, form our network. Choosing the appropriate subset is the key algorithmic problem.
3. Provisioning capacity beyond 1 Gbps along any link involves building additional tower-level links, e.g., by identifying and using links that are also nearly shortest paths, but were omitted in step 1 above.

Step 1: feasible hops. We first use line-of-sight and range constraints to decide which tower pairs can be connected. Achievable tower-to-tower hop length is limited primarily by the Earth’s curvature, which can be treated as a “bulge” of height h_{Earth} . MW hops must clear this curvature and any obstructions in an ellipsoidal region between the sender and the receiver antennae known as the *Fresnel zone*, which has width h_{Fres} . At the midpoint of a hop of length D , using a MW frequency f , we have the following.

$$h_{\text{Fres}} \simeq 8.7m \left(\frac{D}{1 \text{ km}} \right)^{1/2} \left(\frac{f}{1 \text{ GHz}} \right)^{-1/2} \quad (1)$$

$$h_{\text{Earth}} \simeq \frac{1 \text{ m}}{50 K} \left(\frac{D}{1 \text{ km}} \right)^2 \quad (2)$$

In Eq. 2, K accounts for atmospheric refraction [62]. Towers should clear the sum of these heights and any other obstructions. In favorable weather, and with adequately large dish antennae, ranges of up to $D \approx 100$ km are achievable with high availability, provided such line-of-sight clearance [79]. As a specific example, the FCC licensing database [28] indicates that McKay Brothers, LLC (a financial industry provider) operated a $D = 96$ km hop from Chicago, IL (lat. 41.88° , lon. -87.62°) to Galien, MI (lat. 41.81° , lon. -86.47°) as part of a 1183 km MW relay. This example shows that multipath interference issues (associated in this case with a traversal over Lake Michigan) are not an impediment to hop viability.

We assess hop feasibility between each pair of towers by using terrain data made available by NASA [66], which includes buildings and ground clutter, and effectively incorporates the height of the tree canopy.¹ We also require a fully clear Fresnel zone, and adopt $K = 1.3$ and $f = 11$ GHz in the above formulae. The hop engineering routines performing these calculations have been tested in practice: specifically, we have previously used them to design line-of-sight networks, at least 4 of which are now deployed, including ultra-low latency

¹This NASA data set combines data from the Shuttle Radar Topography Mission (SRTM) [66] and the National Elevation Database (NED) [88], and typically yields acceptably small error (~ 2 m) against reference, high-accuracy LiDAR measurements.

routes between data centers hosting financial market matching engines. The methodology routinely provided correct clearance assessments when the physical paths were flashed (confirming line-of-sight with an on-site visit, e.g., [33]). It is relatively rare that the hop feasibility assessment is inaccurate; if a problem arises, it is most likely that the locations themselves are not available to rent. For this reason, in §5.2, we explore relaxations of our tower rental assumptions.

After identifying feasible tower-to-tower hops, for each pair of sites, we find the shortest path through a graph containing these hops, which we call a *link*. In line with observations from the tower data around major population centers, we assume each site itself hosts enough towers to use as the starting point for connectivity from that site to many others.

Step 2: topology design. We need to select a subset of site-to-site links to form a nationwide network that minimizes latency, given a limited budget to spend on links. The Steiner-tree problem [41] can be easily reduced to this problem, thereby establishing hardness. Standard approximation algorithms, like linear program relaxation and rounding, yield sub-optimal solutions, which although provably within constant factors of optimal, are insufficient in practice for this setting. Unfortunately, as we show in Appendix A, solving an Integer Linear program “unsplittable flow” formulation is intractable at the scales of interest. We thus propose two heuristics, the combination of which overcomes the scalability challenge, without substantially deteriorating solution quality.

The first observation we make is that the ILP formulation considers some flow variables that will never take non-zero values, allowing us to eliminate them and any resulting null constraints. For instance, if between two end points, a candidate microwave path is of higher latency than a fiber path (which we can always use, at negligible-in-comparison expense), then it will never carry any flow between these two end points. Similar observations apply to individual “distant, off-path” fiber and MW links. This simple observation substantially reduces the problem size. Note that standard network design problems do not typically have this structure available. This is entirely due to the hybrid design using fiber, which is assumed to be cheap, where available. We benefit, in this case, from having an “oracle” that tells us *a priori* when certain flow assignments are “obviously bad” and will not be useful. Further, carefully defined, such constraints preserve optimality; this part of our solution is not an approximation.

Second, we use a fast greedy heuristic to prune out MW links that are unlikely to be chosen. The heuristic operates using a larger budget ($2\times$ in our implementation) than we are ultimately allowed. In each iteration, we add to the solution the MW site-to-site link that decreases average stretch the most, continuing until the total cost reaches the inflated budget; the chosen links are candidates given to the ILP. Intuitively, the other links are uninteresting – they are unlikely to be picked in the final optimization even when a substantially larger budget is available, and so are not presented as options

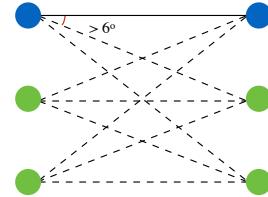


Fig. 1: k^2 bandwidth with $O(k)$ new towers.

to the ILP. This approach does not provide any guarantees, but we find that on small problem sizes, where the exact ILP can also be evaluated, it obtains the optimal solution.

Step 3: capacity augmentation. In many scenarios, some links require more capacity than a single MW connection. For short distances, this is a non-issue: the MW link can simply be replaced by fiber without a large impact on the network’s latency. However, for longer distances, this is not acceptable.

One approach to resolving this problem is simply to build multiple parallel MW links, over multiple series of towers. While tower siting is often a challenging practical problem, with individual sites valued by the HFT industry at as much as \$14 million [59], in the cISP context there is a much larger “tolerance” than in HFT, where firms compete for fractions of microseconds. For a 500 km long cISP link, the midpoint diverging 10 km from the geodesic would increase latency by a negligible 0.2%. Thus, the problem of tower siting is substantially simpler. Also, in many cases, tower infrastructure is dense enough already to allow multiple parallel links. For instance, the HFT industry operates nearly 20 parallel networks in the New York-Chicago corridor [55].

We can also employ a simple trick to enhance the effectiveness of parallel series of towers, as shown in Fig. 1. Instead of k parallel series of towers providing merely a $k \times$ bandwidth improvement, connecting multiple antennae on each tower to other towers, we can obtain a $k^2 \times$ improvement. Using antennae with overlapping frequencies requires an angular separation of 6° [62], as shown in Fig. 1. Again, the stretch caused by the resulting gap between parallel series of towers is small. For a tower-tower hop distance of 100 km, the minimum distance between two parallel towers should be $100 \cdot \tan(6^\circ) = 10.6$ km, which, as noted above, has a small effect on end-to-end latency for long links.

This approach implies that for site-to-site bandwidths under 1 Gbps, we need just one series of towers; for bandwidths between 1-4 Gbps, we need 2 series; for 4-9 Gbps, 3; etc. While tower siting circumstances are often unique, we are aided by two observations: (a) there is substantial redundancy in existing tower infrastructure, and we can often find existing towers for parallel connections (see Fig. 3b and the related text in §4); and (b) when new towers are needed, there is substantial tolerance in where they are sited, as noted above. Bandwidth may potentially be increased even further through spatial diversity techniques, whereby multiple antennae are placed appropriately on the same tower such that they can adaptively cancel

interference by multiple transmission streams within the same frequency channel [89].

4 A cISP FOR THE UNITED STATES

We now apply the framework above for a concrete instantiation: designing a cISP for the U.S. mainland. To assess line-of-sight connectivity between existing towers, we use fine-grained data on tower infrastructure, buildings, terrain, and tree canopy. The fiber conduit data is available from past work [34].

Defining the sites and traffic model: To maximize utility while keeping costs low, we connect only the 200 most populous cities in the contiguous United States. In addition, we coalesce suburbs and cities within 50 km of each other, ending up with 120 population centers. (Henceforth, when we refer to “cities”, we refer to these population centers.) Based on population data for 2010 [20], we calculate that 85% of the US population lives within 100 km of these 120 cities. For the traffic matrix, we use demands between city pairs that are proportional to their population product.

Step 1: Which city-city links are feasible? We use existing towers listed in FCC’s Antenna Structure Registration [39] and databases from American Tower, Crown Castle, and several other tower companies for which we were able to download data. We cull these rather large databases of MW towers to a subset of 12,080 towers as follows: Towers from rental companies are typically suitable for use. From the FCC database, we only use towers over 100 m height. When tower-density exceeds 50 towers per 0.5° square grid cell, we randomly sample towers. (Using all towers could only improve our results, but increases compute time.)

Evaluating link feasibility across tower pairs within range of each other using the aforementioned NASA data [66], we find 261,019 tower-tower hops that satisfy line-of-sight constraints. We find that each city itself has large numbers of suitable towers in its vicinity. We run a shortest path computation on a graph comprising the cities and towers and city-tower and tower-tower hops to find the shortest city-city MW links. This yields both the cost (i.e., number of towers) and latency (i.e., distance along the chosen series of towers) for each city-city link.

For fiber distances, we compute the shortest paths over the InterTubes [34] dataset on US fiber conduits.

Step 2: What subset of links should we build? We use the Gurobi solver [42] to solve our topology design problem. As detailed in Appendix A: (a) both the exact ILP and an LP relaxation approach are too computationally inefficient, while our cISP design heuristic is able to solve the problem at the full scale; and (b) at small scales, where we can also run the exact ILP, our heuristic yields the optimal result.

Fig. 2 shows an example network. Designed with a budget of 3,000 towers and maximum hop length of 100 Km, its average latency is $1.05 \times c$ -latency. Fig. 3a shows the reduction of the network’s stretch with increases in budget for maximum

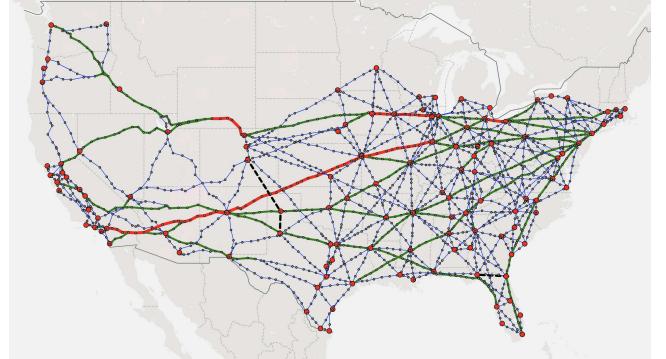


Fig. 2: A 100 Gbps, $1.05 \times$ stretch network across 120 cities in the US. Blue links (thin) need no additional towers. Green (thicker) and red links (thickest) need 1 and 2 series of additional towers respectively. Black dashed links are fiber.

hop lengths of 70 and 100 Km. Given the similarities with 70 and 100 Km, hereon, we only present results for the latter. An animation, showing how the network structure evolves from mostly-fiber to mostly-MW as the budget increases, is available online [26].

Step 3: Augmenting capacity. We produce a target aggregate demand (i.e., the sum of all site-site traffic demands) by scaling our traffic matrix. Then, each tower-tower MW hop that would be over-utilized (given shortest-path routing and the 1 Gbps capacity from §2) is augmented with additional towers at each end, as described in §3. Fig. 2’s topology, when provisioned for an aggregate throughput of 100 Gbps, has 1,660 tower-tower hops that use only already-built towers seen in tower databases, while 552 hops need one additional new tower at each end, and 86 hops need 2 additional towers at each end. Using the cost model described in §2, we find that the cost per GB for this topology, with latency within $1.05 \times$ and 100 Gbps throughput, is \$0.81. For some context, this is $\sim 10 \times$ the cost per GB for content delivery networks [64].

Provisioning even more bandwidth would require more new towers. For 1 Tbps, some tower-tower hops would need as many as 8 additional towers at each end. This is not infeasible — latency would not be inflated excessively, and towers could be found or built. In fact, for the long red link in the map in Fig. 2, which spans 2,700 km from Illinois to California, we find that the *longest* of these 8 additional series of towers would be only 5% longer than the shortest MW path, incurring a stretch of 1.07, instead of 1.02.

We can extend this argument even further: for the same Illinois to California link, we compute tower-disjoint shortest paths, i.e., after finding the shortest path, we remove all towers used by it, find the next-shortest tower-path, etc. In this process, we use only existing towers from our databases, and adhere to the same link feasibility constraints. Fig. 3b shows that stretch increases gradually as we keep eliminating towers; nevertheless, even after 20 such iterations, stretch is much smaller (1.15) than with the existing fiber conduit

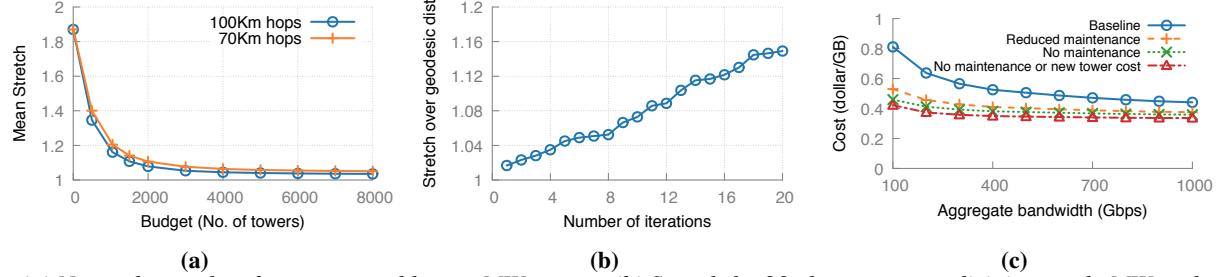


Fig. 3: (a) Network stretch reduces as we add more MW towers. (b) Stretch for 20 shortest tower-disjoint purely MW paths along the long red IL-CA link in Fig. 2. (c) Cost per GB for the city-city traffic model decreases with increasing aggregate throughput.

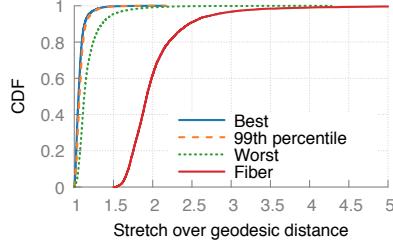


Fig. 4: Stretch across all city-pairs over a year of weather. The 99th-percentile stretch is comparable to the best stretch.

(1.75). Note that this route runs through the Rocky mountains and other areas of low tower density. Thus, in accounting for the cost of bandwidth augmentation entirely using the (higher) cost estimates for building new towers, we are substantially overestimating the expense.

There is also another reason our costs are over-estimates: at sufficiently high bandwidth, there is a better option than building many parallel long-distance MW links: one could use the same number of towers to construct a single line of towers with shorter tower-tower distances. This can make shorter-range, but higher-bandwidth technologies like MMW or free-space optics, more cost-effective.

Despite the above two factors, we use parallel MW towers, with all the required additional towers accounted for as new towers, to provide conservative cost-estimates as aggregate bandwidth increases in Fig. 3c.

Routing, queuing, and traffic models. We show in Appendix B that: (a) routing that incurs small (under 10%) latency inflation compared to shortest paths can drive the network at virtually zero loss and minimal queuing delay even at high utilization; and (b) packet pacing addresses the problem of edge links having higher line rates than cISP links. We also discuss evidence for per-MW-hop latency overheads being small enough to ignore. Further, in Appendix C, we show that besides the population-product model, cISP can also be tailored for inter data center traffic, data center to edge traffic, and various combinations of these.

Alternative deployment models. The deployment model and analysis have been conservative in assuming high maintenance and tower installation costs for the provider. What if an incumbent tower company like American Tower [7] deployed cISP? (Fig. 14 in the Appendix shows that American

Tower’s existing deployment broadly covers areas where our network design of Fig. 2 requires towers.) Besides reduced tower installation costs, maintenance would also be significantly reduced due to the obligation to maintain towers for customers anyway. We evaluated several scenarios of this type, as shown in Fig. 3c. While the solid line represents the baseline deployment model discussed in §2, the dashed lines represent models with reduced maintenance cost (\$10K per tower per year), no maintenance cost, and no maintenance or new tower cost (only antenna cost). A network with 3,000 towers offering 100 Gbps bandwidth and 1.05 stretch, built by a company like American Tower, could cost as little as \$0.42/GB, thus reducing the baseline cost by almost 50%.

Finally, we note that cISP could be deployed in other geographies besides the US. As discussed in Appendix C.3, we could design a cISP for Europe offering a stretch of 1.04 (vs. 1.05 for the U.S.) with a budget of $\sim 3k$ towers.

5 PRACTICAL CHALLENGES

Deploying cISP would involve several practical challenges beyond network design and routing, which we now address.

5.1 Impairments due to weather

We use standard equations from MW engineering [48] to calculate signal attenuation due to precipitation. We assume hardware characteristics of a standard low-latency MW radio: an 8-foot dish with a gain of 46.5 dBi at 11 GHz [29, 74, 75]. While antenna gain is determined by the hardware, transmit power and receive power thresholds also depend on the modulation scheme (256 QAM). Following ITU models [48], at ~ 11 GHz, precipitation is likely to be the dominant source of attenuation. While the physical layer could trade link bandwidth for higher resilience to weather, we treat the impact of precipitation in a binary manner: if attenuation exceeds a threshold that would degrade bandwidth, we conservatively consider a link to have failed.

We assume that when a link fails, traffic is shifted to the shortest available route, which may use any combination of MW and fiber. The high precipitation that causes failures is easy to predict, especially on the timescale of minutes. Thus, even slow, centralized management would suffice to anticipate failures and reroute accordingly.

We use NASA’s precipitation data [65] to determine which links are down when, and what the impact of such failures

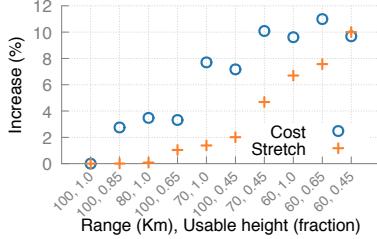


Fig. 5: As constraints on tower space and range become tighter, the network becomes more expensive, and stretch increases.

is on the network’s latency. For each day over a period of a year (July 2015 - June 2016), we select a 30-minute interval uniformly at random, and identify the links that would fail during it. We then evaluate the latency for each pair of cities end-to-end for each interval. Fig. 4 shows that 99th-percentile latencies are nearly the same as the best fair-weather latencies. In terms of the median across city-pairs, even the worst latencies over the year are 1.7 times lower than those over fiber. Large increases in latency due to weather typically occur only between nearby city-pairs, the fiber route to which runs through a farther-away city, e.g., in Texas, Austin and Killeen fall back to a fiber route through Fort Worth. A more sophisticated analysis allowing dynamic link bandwidth adjustment rather than binary failures can only improve these numbers. Thus, even under significantly adverse weather, most of the latency advantage of cISP remains intact.

We have also created an animated visualization of the network’s latency evolving over a year’s weather [25].

5.2 Tower height and availability

Our initial design assumed a MW hop to be feasible if it spans a distance of 100 km or less, and satisfies line-of-sight constraints using the tops of the towers. In practice, however, a tower chosen for a route might not have a free spot for a new antenna at the necessary height, especially at the top, where structural concerns for large parabolic antennae are greatest, and where access and maintenance can be problematic. Further, for smaller antennas, insufficient gain margins can decrease the 100 km maximum range. Hence, we evaluate cost and latency of the network with hop-level restrictions modeling these effects.

We test the impact of restricting usable height on towers to three levels, as a fraction of tower height: 0.85, 0.65, and 0.45. Testing for line-of-sight visibility with these restrictions eliminates more towers than using tower tops. We also vary the maximum range, which can necessitate the use of a larger number of towers, thus increasing the cost and potentially making some city-pairs infeasible to connect using MW.

We assess the *percentage increase* in cost and stretch values compared to the baseline values with 100 km range and using the tower tops, i.e., height fraction = 1. Fig. 5 shows the results for different combinations of the range and antenna-height constraints, sorted by lowest to highest stretch. The

maximum increase in cost is 11% (with the absolute cost per GB under these constraints being \$0.90), while the maximum increase in stretch is 10% (with the absolute stretch compared to the geodesic being 1.16). Thus, even substantial potential problems with mounting antennas do not change our overall conclusions about the viability of cISP.

In our experience designing MW routes, assessments like the ones in this work have yielded accurate estimates of the latency and the number of tower-tower hops that will ultimately be used to connect two sites. The precise set of towers often differs based on real-world constraints, particularly tower unavailability for structural and rental-related reasons. Thus, while accurate in terms of cost and latency, this work does not provide fully engineered routes. In practice, to improve accuracy in preparation for building a MW route, we assign an acquisition probability to each tower in a swath connecting the sites, which depends on a number of factors (e.g., tower type, ownership, and location). Further, for towers that can be acquired, we use a uniform distribution to model height at which space for antennas is available. With this probabilistic model, we compute thousands of candidate MW paths between site pairs, with refinements as acquisitions and height availabilities are confirmed. We make available in video form [24] an example of such refinement.

5.3 Integration into the Internet

We next discuss potential problems cISP may face in terms of integration into the present Internet ecosystem.

Low-hanging fruit: The easiest deployment scenarios involve one entity operating a significant network backbone:

- A CDN could use cISP to carry “back-office” traffic between its locations and content origins, which often supports latency-sensitive user-facing interactions [73]. While the strategies of moving content closer to end users and speeding up the network are orthogonal, on cache misses and when serving uncachable content, only speeding up the network improves performance.
- Content-providers like Google and Facebook can use cISP to carry latency-sensitive traffic – such WAN designs already accommodate distinctions between such traffic and background traffic [47, 50].
- Purpose-built networks such as for gaming [40] can easily use cISP between their edge locations and servers.

All of these are interesting and economically viable use cases with minimal deployment barriers, and each *alone* may justify a design like cISP. For instance, while it is tempting to dismiss gaming as a niche, it is a large and growing market: the Steam gaming platform claims 20+ million players worldwide [85]. At a 10 Kbps rate per player [27], this aggregates to 27 Gbps – enough to make cISP viable in this setting. (We present cost-benefit estimates, including for gaming, in §8.)

User-facing deployment: Access ISPs may use cISP as an additional provider, and incorporate a low-latency service

into their broadband plans.² Utilizing cISP in this manner can help ISPs to provide and meet the requirements of demanding Service Level Agreements, the case for which was made in recent work [14]. ISPs may use heuristics to classify latency-sensitive traffic and transit it using cISP. Alternatively, software at the user-side may make more informed decisions about which traffic should use the fast-path exposed by the ISP. While this would require significant user-side changes, note that many of today’s applications already manage multi-modal WiFi and cellular connectivity.

6 EMPIRICAL RESULTS

To evaluate the characteristics of long-haul microwave links, we have conducted experiments over one of the most popular nearly-speed-of-light networks deployed in the high-frequency trading corridor between Chicago and New Jersey. We describe these experiments and their results below. The HFT niche is partially characterized by a “winner-takes-all” dynamic which requires these networks to operate at the bleeding edge of low latency. Hence, it is important to quantify the usefulness of these networks in serving more generic low-latency applications on the Internet, which have less-strict latency requirements than HFT, but higher availability and lower packet loss demands.

6.1 Active measurements

We conducted active measurements over the microwave link between the Chicago Mercantile Exchange (CME) data center and the Equinix data center in Secaucus, New Jersey, operated by one of the fastest MW networks in the corridor. On weekdays, when the Chicago and New York markets are open, the link carries financial information critical to high-frequency trading that triggers trades worth billions of dollars. The networks are optimized for low latency, with microseconds of advantage [13] providing a significant edge to customers.

We ran experiments for ~ 7 hours every Saturday for 11 weeks between Nov. 2019, and Oct. 2020 from one host each located in the CME and Equinix data centers. The microwave link was provided to us without any Forward Error Correction (FEC), thus being exposed to all errors and bit flips expected in radio transmission. We observe that the link behavior tends to be in one of two states: losses are either very low (normal) or very high (degraded). Out of a total of 72 hours of measurements, there are 12 hours during which the link is degraded due to weather, and 4 hours during which it is down due to maintenance or other issues. Note that because there is no FEC at all, very small bit error rates (BER) degrade the link. Also, in our trading data analysis (§6.2), we see that microwave networks stay up in worse weather conditions than these 12 hours. FEC is needed in packet headers to correct for bit errors, which we could not implement as we did not have access to routers on the network.

²While large last-mile latencies can overshadow cISP’s low latency, this is an entirely orthogonal problem, on which significant progress is being made – 5G prototypes are already showing off sub-millisecond latencies [46].

6.1.1 RTT and bandwidth

The geodesic distance between the CME and Equinix data centers is 1139.5 km. The c -latency for a round-trip, then, is 7.6 ms. In our experiments over 11 weeks, we always observe a round-trip time of 7.7 ms for 32-byte packets, i.e., within 1.5% of c -latency. The RTT goes up to 7.9 ms for 1,499-byte packets because of the limited bandwidth available on the link (or more specifically, the slice of it provided to us).

The 0.1 ms increase in transmission delay as packet size increases by 1,467 bytes gives a bandwidth estimate of 120 Mbps. Our UDP measurements and TCP measurements, in the best case, also give us a bandwidth of 120 Mbps. It is hard for TCP to sustain throughput at this rate in the absence of any FEC because of transmission losses. While the operator did not divulge the exact link capacity, it is likely that our network access was capacity-capped. Hence, these measurements only provide a lower bound on the link bandwidth.

6.1.2 Loss and FEC

In plain TCP (iperf) and ICMP (ping) probes, we observe high loss rates: typically around 3% to 5% for 32-byte packets. The packet loss rate increases sharply as packet size increases because more bits can potentially be corrupted in transmission. Without FEC, a link with loss rate this high is clearly unsuitable for web traffic [91]. Whether FEC can bring the loss rate down to an acceptable level (say, 0.1%) at reasonable latency and bandwidth overhead depends on two factors: 1. the Bit Error Rate (BER), and 2. the typical length of error bursts, i.e., how many consecutive bits are corrupted in an error burst. We elaborate on these factors below.

First, we derive the underlying BER from observed ping packet loss. For a ping packet of s bytes, a successful response is observed when both the echo request and reply packets are delivered to the respective hosts without any errors. To estimate the BER b_{err} , we first assume that bit errors are uniform and random. Then, for packet loss rate p_{loss} , we get:

$$b_{err} = 1 - (1 - p_{loss})^{1/(2 \times 8 \times s)}$$

For initial validation of this model, with the possibly unjustified assumption of random and uniform errors, we calculate b_{err} from observed p_{loss} for $s = 1,499$ for the 7 hours of measurements on Feb. 15th, 2020. Then, we use the calculated b_{err} to predict p_{loss} for $s = 396$ on the same day. We compare the predicted and observed values in Fig. 6a. While the observed and predicted loss rates for $s = 396$ largely agree, there are some disagreements, e.g., at 12:30, which can be explained by the fact that the observations for $s = 1,499$ and $s = 396$ are separated in time by 60 seconds. The underlying BER might change during this interval. For Feb. 15th, the median, 95th percentile, and maximum BER we calculate are 3.6×10^{-5} , 8.2×10^{-5} , and 3.6×10^{-4} respectively.

For a target packet loss rate of 0.1% for packets of size 1,500 bytes, the BER needs to be 4.17×10^{-8} or lower. Extremely lightweight FEC codes, such as Reed-Solomon (255,

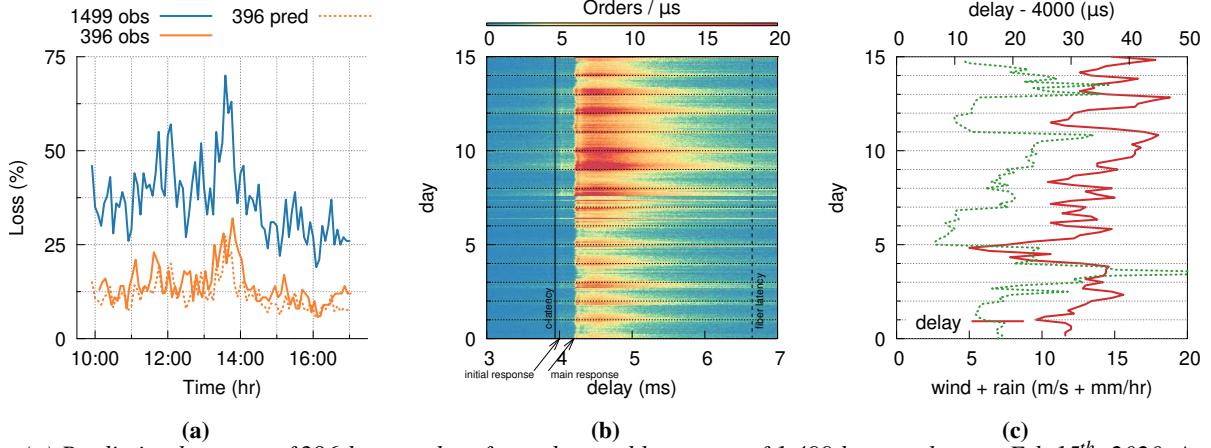


Fig. 6: (a) Predicting loss rate of 396 byte packets from observed loss rates of 1,499 byte packets on Feb 15th, 2020. Analyzing trading data: (b) Heat map of order book events at delay between Chicago and New Jersey. Response delay never exceeds 4.3 ms; (c) A coarse weather signal (max wind speed + max rainfall) is correlated with the observed transmission delay.

239) can correct from BER of 10^{-4} to 10^{-12} with a bit rate overhead of only 7% [76]. If performed over 255 byte blocks, a 1,500 byte packet can be encoded in 7 blocks with a total redundancy overhead of 112 bytes. At 120 Mbps bandwidth, this incurs a latency penalty of only 7.5 μ s. This FEC scheme would break down, however, if errors occurred in bursts of around 8 bytes or more. Now we discuss the earlier assumption of error bursts being short and uniformly distributed.

To analyze bit errors, we sent two sets of UDP probes over the link: the first set consists of 60 byte packets sent at 35 packets per second (slow), and the second consists of 60 byte packets sent at 200,000 packets per second (fast). The slow set characterizes link behavior with no congestion/bandwidth related losses, whereas the fast set provides statistical significance to rare bit flip events. In contrast to ping losses, losses in this experiment are observed through packet captures rather than at the application layer, so a corruption of, e.g., the UDP destination port would not register a loss. For the slow set, we observe a packet loss rate of 0.8%, whereas for the fast set we observe a loss rate of 2.04%.

In the UDP fast set a packet has 4 bytes of payload, 8 bytes of UDP header, 20 bytes of IP header, 14 bytes of Ethernet header, and 14 bytes of padding. A total of 1.6 billion packets were sent, out of which 2.66 million were received on the other end with at least one of the following fields corrupted: source port, destination port, UDP header length field, and payload. We calculate the Hamming distance between the received value and the expected value of the corrupted fields. As Table 7a shows, there appears to be a linear relationship between field size and number of corruptions, and over 99% of all corruptions consist of 2 bit flips or less. Also, if we extrapolate the errors we observe in these 4 fields to the rest of the 60 byte packet, the expected loss rate due to corruptions in the Ethernet and IP headers and padding matches that observed in the UDP slow set. The other 1.24% packets lost can thus be explained by congestion/bandwidth issues.

6.2 Trading data analysis

To characterize the latency and up-time of the full range of microwave links deployed in the Chicago-New Jersey corridor, we analyze trading data from the Chicago Mercantile Exchange (CME) in Chicago, Illinois, and the CBOE Options Exchange in Secaucus, New Jersey. Information about trades happening at the CME travels over microwave paths and triggers activity at the CBOE [13]. The time difference between stimulus events at the CME and the response at the CBOE represents the network latency between the two exchanges. Laughlin et al. have also used this methodology to estimate latency between financial markets [55].

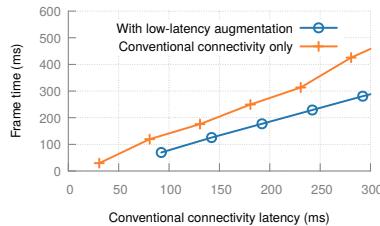
We obtained tick data from CME and CBOE for three weeks of Mar. 2019. The tick data consists of microsecond precision timestamps for events at both ends. Both markets are open simultaneously for 6.5 hours every weekday, which means that we have 97.5 hours of relevant tick data. For each trade executed at the CME at timestamp t , we count the number of order book events at the CBOE at timestamps $t+i$ where $i \in [3000, 7000]$ μ s. Fig. 6b plots a heat map of the number of orders per μ s for each 10 μ s bin in the tick data. The y-axis time is in intervals of 15 minutes. Analysis of the data shows that the main response delay, which reflects the network latency between CME and CBOE, does not exceed 4.3 ms for any 15-minute interval. The lowest fiber latency between the two exchanges is 6.65 ms [60]. This shows that some microwave networks were up through every 15-minute interval over the 3-week period.

In addition to the main response at 4.2 ms, Fig. 6b has a smaller initial response at 4.0 ms. The CME tick data reveals that internal trading algorithms and strategies produce a second stimulus at CME 200 μ s after the initial stimulus. The main response in Fig. 6b is triggered by that second stimulus.

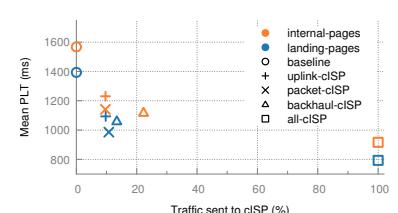
We consider the delay between the second stimulus at CME and the main response at CBOE as transmission delay. We calculate the transmission delay for every 1-hour interval

Field	#corruptions	#bits	1 bit flip	2 bit flips
src port	873,165	16	84%	15%
dst port	864,955	16	82%	17%
length	914,528	16	85%	14%
payload	1,734,539	32	84%	15%

(a)



(b)



(c)

Fig. 7: (a) Corruptions observed in the UDP fast set. (b) A substantial reduction in frame time can be obtained by the use of a parallel low-latency augmentation to the present Internet. (c) Mean web page load time (PLT) improvement for each heuristic and its portion of traffic delivered on cISP. PLT can improve substantially by only offloading a small portion of traffic to cISP.

in the tick data. Fig. 6c plots the moving average of transmission delay over 2 hours. We use the hourly wind speed estimate [30] and rainfall data [22] in the regions through which the MW corridor passes as a coarse weather signal. For each hour, we pick the maximum wind speed and maximum rainfall observed at a granularity of $\sim 10\text{ km}$ along the geodesic between the end points. Fig. 6c plots wind speed + rainfall /2, and shows that there is some correlation. The Pearson correlation coefficient between wind and delay is 0.24, while that between rain and delay is 0.16. Sources of noise in this correlation include the noise inherent in the trading data itself, and issues that may affect transmission delay, such as infrastructure damage or operational downtime. Note that days 3 and 14 have more severe rain and wind than the 12 hours during which the link was degraded in our active measurements (§6.1).

Conclusions: From the active measurements, we conclude that for our MW path, (1) round-trip latency is less than 1.5% inflated over c -latency, (2) bandwidth is at least 120 Mbps, (3) error bursts are very short and roughly uniformly distributed under normal link conditions, and (4) errors can be brought down to acceptable levels with extremely lightweight FEC incurring minimal latency and bandwidth overhead.

From the trading data analysis, we conclude that (1) for the 97.5-hour period, some MW networks, spanning more than 1,000 km, were always up without any significant degradation in latency, and (2) weather events such as high wind speeds and rainfall are correlated with increases in transmission delay by tens of microseconds. This increase may stem from one or more of the following: (a) longer end-to-end paths being picked, (b) shorter tower-to-tower hops leading to higher switching delay, and (c) the network responding to weather changes by ramping up FEC.

7 A FEW POTENTIAL APPLICATIONS

Several applications require low latency over the wide area-network. Applications focused on user interactivity, such as augmented and virtual reality, tele-presence and tele-surgery, musical collaboration over long-distances, etc., can all benefit from low-latency connectivity. Likewise, less user-centric applications, such as real-time bidding for Web page adver-

tisements [8] and block propagation in blockchains, would also benefit. While it is beyond the scope of this paper to analyze this in detail, we assess, in simplified environments, the improvements cISP could achieve for two applications.

7.1 Online gaming

We discuss cISP’s benefits for both models of online gaming: **thin-client** (where a client essentially streams everything in real-time from a server) and **fat-client** (where the client has the game installed, performs computations, etc., and only relies on the server for updates on the global game state).

Fat-clients are dominant today, and are easy to tackle: communication is almost entirely composed of latency-sensitive player actions and game-state changes, and is low-volume, typically a few Kbps per client for popular games [27]. It can all be transferred over the low-latency network, reducing latency by $3\text{-}4\times$ compared to today’s Internet.

Thin-client gaming is still in its infancy, as it depends heavily on the network, with data rates in Mbps. We explore the potential of a speculative approach: the server speculates on the game state and sends data for multiple scenarios in advance over fiber, then on the low-latency network, issues messages indicating which scenario occurred. Such speculation has already shown success for rich games like “Doom 3” [56].

We use a toy thin-client for a multi-player Pacman variant to explore the latency benefit. Our rudimentary implementation speculates on all 4 movement directions possible as user input. In line with the online-gaming literature, we measure “frame-time,” which “corresponds to the delay between a user’s input and the observed output” [56]. We evaluate frame-time as latency over conventional connectivity increases (emulated by adding latency in software), and for a low-latency network always incurring $1/3$ of the latency of the corresponding conventional network.

As Fig. 7b shows, the speculative approach enabled by the low-latency network augmentation reduces frame-time. This comparison would improve further if non-network overheads from processing and rendering in our naive implementation were smaller. We do not use any heavy graphics on which to evaluate the additional bandwidth overhead on fiber, but even in the sophisticated scenarios examined by prior work [56], this bandwidth overhead can be contained to $2\text{-}4.5\times$.

7.2 Web Browsing

We evaluate the potential impact of cISP’s latency improvement on Web page load times (PLTs) (based on the *onLoad* event [71]) using Mahimahi [68] with the addition of content delivery network (CDN) caching. Our emulation supports two levels of the CDN cache hierarchy. The client’s request first reaches the edge server. If it is a cache miss, the request will be forwarded to the parent server. In case there is another miss, it will be forwarded to the origin server. This setup thus allows variable request latency, where certain requests can experience more latency.

To realistically recreate the caching behavior, our experiments leverage the Akamai pragma header [3] which is typically used for debugging purposes. We select web pages where at least 75% of the HTTP requests³, performed when loading a page, are served by Akamai. Overall, we found 27 landing pages and 140 associated internal pages from the Hispar list [19] match this criterion. We record each page’s content and the network latency for each (edge) server that a client contacted when loading a page. This recording process is conducted from three different vantage points at three different times. For the CDN server-to-server latency, we estimate the latency by geolocating the IP addresses of the CDNs and origin servers provided by the pragma header⁴. We then replay each page with unmodified network latencies (as a baseline) and with latencies reduced to $0.33 \times$ of their original values (as a cISP). No bandwidth limitations are imposed.

Fig. 7c shows the results. Compared to the baseline, a 66% reduction in latencies (all-cISP) results in a mean 42% PLT decrease for both landing and internal pages (an absolute decrease of 600 ms and 651 ms). This PLT reduction is less than the 66% reduction in RTT because loading a Web page also involves significant non-network activity.

If cISP is used only to deliver the CDN’s server-to-server (i.e., back-office) traffic, our experiment (backhaul-cISP) suggests that PLT can be improved by 23.7% and 28.5% (331 ms and 447 ms) on landing and internal pages by only sending 13.4% and 22.3% of the overall web-browsing traffic on cISP. Internal pages get better improvement and send a higher proportion of traffic because they experience more cache misses (31.9%) compared to landing pages (13.3%).

While Web-browsing traffic comprises only a small fraction of total Internet traffic⁵, we can further reduce the load by carrying only *latency-sensitive* traffic on cISP. Hence, we extend Mahimahi to enable *selective* manipulation of RTTs in the replay, such that some traffic sees lower RTTs than other traffic. We test two heuristics under this setup. First, we try a simple heuristic that only sends uplink traffic to cISP (uplink-cISP). This approach yields a mean PLT im-

provement of 21.5% (319 ms) by sending only 9.7% of the web-browsing traffic over cISP. Second, we adopt a more advanced PKT-State heuristic [77] (packet-cISP) to distinguish the latency-sensitive traffic (e.g., TCP SYN/ACK packets and small data packets) from the bandwidth-intensive traffic (e.g., data packets). By offloading the latency-sensitive traffic to cISP, we can get a mean PLT improvement of 28.2% (417 ms) by only offloading 10.2% of the traffic.

8 COST-BENEFIT AND MARKET ANALYSIS

Does cISP’s value justify its cost? For three important use cases, we present quantitative lower-bound estimates of cISP’s value per GB. cISP would also need enough aggregate demand across one or more use cases to support its total deployment cost, so we estimate market size of each use case.

Web search. *Value per GB:* Putting together Google’s quantification of the impact of latency in search [18], their estimated search revenue restricted to the US [63], their search volume [84], estimated data transferred per search⁶, and estimated cost per search [53], we estimate that speeding up page load times for 12 Gbps of their US search traffic by only 200 ms (400 ms) would yield an additional yearly profit of \$87 (\$177) million. This translates to an added value of \$1.84 (\$3.74) per GB. *Market size:* At 12 Gbps of traffic, Google’s search traffic is a nontrivial fraction ($> 10\%$) of a cISP provisioned to provide ~ 100 Gbps, but to make cISP viable, it would have to be augmented with other use cases.

E-commerce. *Value per GB:* Using Amazon.com’s estimates of number of visits, pages fetched per visit, fraction of US traffic [80], and page size, we arrive at an estimated 480 PB of US traffic per year. Using their US sales [32] and profit margin of 5.5% [61] gives an estimated \$16.3 billion in profits per year. Estimates for the effect of PLT on conversion rate vary from 1% [57] to 2.4% (on desktop) and 7% (on mobile) per 100 ms of additional latency [5]. Thus, saving 200 ms by sending only 10% of the data over cISP (\$7.2), translates to a value of \$6.8-\$47.5 per GB, which is much higher than the \$0.81 per GB cost of cISP traffic. *Market size:* 10% of 480 PB of Amazon e-commerce annual US traffic translates to 12 Gbps of cISP traffic. But the current (2020) e-Commerce market size of \$861 billion [32] (compared to Amazon’s $\sim \$296$ billion) proportionately translates to a cISP traffic demand of 35 Gbps. Given the high value per GB, this use case alone could make a 100 Gbps cISP profitable.

Gaming. *Value per GB:* Online gamers often pay for “accelerated VPNs”, which promise to lower network latency. Such services cost \$4-\$10 per client per month [1, 11, 72]. Full-time gaming at 8 hours a day at a 10 Kbps rate (as in §5.3) translates to 1.08 GB / month. Thus, if cISP were priced like a cheap accelerated VPN service at \$4 / mo, this would translate to a value of at least \$3.7 / GB. A less aggressive model than “full-time gaming” would only improve cISP’s value. Another indicator of latency’s value in gaming is the

³We assume requests not served by Akamai are served by the edge server.

⁴We geolocate each server, and compute server-to-server c-latency from distance. Then, we estimate baseline latency as $3 \times c$ -latency.

⁵Cisco’s 2018 estimate puts “Web/Data traffic” at 13% [23] including non-latency sensitive traffic like software updates and some file transfers.

⁶From Firefox desktop’s network tools; mobile responses may be smaller.

market for gaming monitors with high screen-refresh rates: the 6-10 ms of latency advantage is valued at over \$50 by many gamers, estimated from the pricing of monitors which are exactly the same except in terms of refresh rate [6]. *Market size:* There are more than 350 million [83] Fortnite gamers worldwide. Assuming 20% of the gamers are in the US, each with a demand of 10 Kbps, translates to 700+ Gbps of cISP demand. Even for games with smaller user bases like PUBG (70 million) and Call of Duty Warzone (100 million), cISP demands are high enough to sustain a nationwide network.

Summary. The value per GB obtained from cISP’s latency reduction in the above cases – \$1.84–\$3.74, \$6.52–\$45.63, and over \$3.70 – exceeds its cost estimate of $\leq \$0.81$ per GB, and even leaves room for substantial over-provisioning. Total addressable market demand could greatly exceed a 100 Gbps cISP for the case of gaming, and for web-based use cases could be sufficient to support the infrastructure.

This simplified analysis omits many factors. Not all users would be paying for the infrastructure on day 1, so an incremental roll-out for a smaller set of customers would be important. Also, there are many other applications that can benefit from cISP. CDNs routinely use overlay routing to cut latency for dynamic, non-cacheable content, for which edge replication is difficult or ineffective [4]. Upcoming application areas like virtual and augmented reality can only make the case stronger for cISP. We expect cISP’s most valuable impact to be in breaking new ground on user interactivity, as explored in some depth in prior work [16].

9 RELATED WORK

Networking research has made significant progress in measuring latency, as well as improving it through transport, routing, and application-layer changes. However, the underlying infrastructural latency has received little attention and has been assumed to be a given. This work proposes a speed-of-light ISP, demonstrating that improvements are indeed possible.

There are several ongoing Internet infrastructure efforts, including X moonshot factory’s project Taara [90], Facebook connectivity’s Magma [36], Rural Access [37], Terragraph [38], and the satellite Internet push by Starlink [81], Kuiper [54], Telesat [86], and others. Project Taara consists of networks under deployment in India and Africa, based on free-space optics, and described as “Expanding global access to fast, affordable internet with beams of light”. While Facebook’s Magma and Rural Access aim to extend connectivity to rural areas by offering a software, hardware, business model, and policy framework, Terragraph aims to extend last-mile connectivity to poorly connected urban and suburbs areas by leveraging short millimeter-wave hops. Free-space networks of this type will likely become more commonplace in the future, and these works are further evidence that many of the concerns with line-of-sight networking can indeed be addressed with careful planning. Further, cISP’s design approach is flexible enough to incorporate a variety of media

(fiber, MW, MMW, free-space optics, etc.) as the technology landscape changes.

“New Space” satellite networks: While low-Earth orbit (LEO) satellite networks can reduce long-distance latency [12, 44, 52], current deployments are more targeted at last-mile connectivity than long haul [15]. Starlink recently claimed to offer last-mile round-trip latency of 31 ms [82], more than $3.8 \times$ the latency estimated in prior simulations [12], showing that the service is not yet latency optimized.

Despite the apparent differences in objectives — long haul latency for cISP and last-mile connectivity for LEO networks — it is useful to **coarsely** assess how the costs may compare. Starlink, for example, offers uncapped connectivity at \$99/month [78]. At an average household consumption of 273.5 GB [35], this translates to \$0.36/GB⁷. For cISP, if an incumbent like American Tower were to deploy it, the cost could be as low as \$0.33/GB, as shown in Fig. 3c. Thus, a network with costs comparable to cISP (in a per-bit sense; cISP is more than an order of magnitude cheaper in absolute cost, and has commensurately lower bandwidth) is concurrently being deployed, albeit with different goals.

To the best of our knowledge, the only efforts primarily focused on wide-area latency reduction through infrastructural improvements are in niches, such as the point-to-point links for financial markets [55], and isolated submarine cable projects aimed at shortening specific Internet routes [67, 69].

10 CONCLUSION

A speed-of-light Internet not only promises significant benefits for present-day applications, but also opens the door to new possibilities, such as *eliminating the perception of wait time* in our interactions over the Internet [16]. We thus present a design approach for building wide-area networks that operate nearly at c -latency. Our solution integrates line-of-sight wireless networking with the Internet’s fiber infrastructure to achieve both low latency and high bandwidth.

A speed-of-light Internet has not always been clearly viable. The enabling technology of low-latency multi-hop microwave networks was spurred on by HFT only within the last 10 years, and even then it has not been a priori obvious that the challenges of relatively high loss and low bandwidth could be overcome to leverage such links for an Internet backbone. More importantly, the Internet has become increasingly latency-limited due to increasing bandwidths and greater use of interactive applications. Thus, we believe we have reached an exciting point in time when greatly reducing the Internet’s infrastructural latency is not only tractable, but surprisingly cost-effective and impactful for applications.

ACKNOWLEDGEMENTS

This work was supported by National Science Foundation Awards CNS-1763492, CNS-1763742, and CNS-1763841.

⁷Starlink is currently in beta testing, and profit margins are unclear. It is difficult to do a tighter cost analysis for Starlink without more information.

REFERENCES

- [1] AAA Internet Publishing, Inc. WTFast. <https://www.wtfast.com/en/>. [Online; accessed 11-March-2021].
- [2] Akamai. Akamai “10for10”. <https://www.akamai.com/us/en/multimedia/documents/brochure/akamai-10for10-brochure.pdf>, July 2015. [Online; accessed 11-March-2021].
- [3] Akamai. Using Akamai Pragma headers to Investigate or Troubleshoot Akamai Content Delivery. https://community.akamai.com/customers/s/article/Using-Akamai-Pragma-headers-to-investigate-or-troubleshoot-Akamai-content-delivery?language=en_US, 2015. [Online; accessed 11-March-2021].
- [4] Akamai. SureRoute. <https://developer.akamai.com/learn/Optimization/SureRoute.html>, 2017. [Online; accessed 11-March-2021].
- [5] Akamai. The State of Online Retail Performance. <https://www.akamai.com/uk/en/multimedia/documents/report/akamai-state-of-online-retail-performance-spring-2017.pdf>, 2017. [Online; accessed 11-March-2021].
- [6] amazon.com. ASUS VG248QE Gaming Monitor. <https://goo.gl/gnFnPv>, 2018. [Online; accessed 11-March-2021].
- [7] American Tower Global Wireless Solutions. <https://www.americantower.com/us/>, 2004. [Online; accessed 11-March-2021].
- [8] Waqar Aqeel, Debopam Bhattacherjee, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. Untangling header bidding lore: Some myths, some truths, and some hope. In *Passive and Active Measurement*, 2020.
- [9] Waqar Aqeel, Balakrishnan Chandrasekaran, Bruce Maggs, and Anja Feldmann. On landing and internal pages: The strange case of Jekyll and Hyde in Internet measurement. In *ACM IMC*, 2020.
- [10] AT&T Corporation. AT&T Long Lines Routes March 1960. <http://long-lines.net/places-routes/maps/MW6003.html>, 2003. [Online; accessed 11-March-2021].
- [11] Battleping. Info on our lower ping service. <http://www.battleping.com/info.php>, 2010. [Online; accessed 11-March-2021].
- [12] Debopam Bhattacherjee, Waqar Aqeel, Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P. Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. Gearing up for the 21st century space race. In *ACM HotNets*, 2018.
- [13] Debopam Bhattacherjee, Waqar Aqeel, Gregory Laughlin, Bruce M. Maggs, and Ankit Singla. A bird’s eye view of the world’s fastest networks. In *ACM IMC*, 2020.
- [14] Zachary S. Bischof, Fabián E. Bustamante, and Rade Stanojevic. The Utility Argument - Making a Case for Broadband SLAs. In *PAM*, 2017.
- [15] Bloomberg. Musk targets telecom for next disruption with Starlink Internet. <https://tinyurl.com/wejrv37c>, 2021. [Online; accessed 11-March-2021].
- [16] Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, Brighten Godfrey, Gregory Laughlin, Bruce M. Maggs, and Ankit Singla. Why Is the Internet so Slow?! In *PAM*, 2017.
- [17] Ilker Nadi Bozkurt, Waqar Aqeel, Debopam Bhattacherjee, Balakrishnan Chandrasekaran, Philip Brighten Godfrey, Gregory Laughlin, Bruce M. Maggs, and Ankit Singla. Dissecting latency in the Internet’s fiber infrastructure, 2018. arXiv:1811.10737.
- [18] Jake Brutlag. Speed Matters for Google Web Search. <http://goo.gl/vJq1lx>, 2009. [Online; accessed 11-March-2021].
- [19] Gustavo Carneiro, Pedro Fortuna, and Manuel Ricardo. FlowMonitor: A Network Monitoring Framework for the Network Simulator 3 (NS-3). In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS ’09, 2009.
- [20] Center for International Earth Science Information Network (CIESIN), Columbia University; United Nations Food and Agriculture Programme (FAO); and Centro Internacional de Agricultura Tropical (CIAT). Gridded Population of the World: Future Estimates (GPWFE). <http://sedac.ciesin.columbia.edu/gpw>, 2005. [Online; accessed 11-March-2021].
- [21] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *USENIX OSDI*, 2014.
- [22] CHRS at UC Irvine. PERSIANN-CCS. <https://chrsdata.eng.uci.edu/>, 2017. [Online; accessed 11-March-2021].
- [23] Cisco. Cisco Visual Networking Index: Forecast and Methodology. <https://www.reinvention.be/webhdfs/v1/docs/complete-white-paper-c11-481360.pdf>, 2017. [Online; accessed 11-March-2021].
- [24] cISP authors. MW path refining. <https://goo.gl/LwYB5Z>. [Online; accessed 11-March-2021].

- [25] cISP authors. Impact of rainfall on cISP for a period of 1 year. <https://tinyurl.com/a8szcukz>, 2021. [Online; accessed 11-March-2021].
- [26] cISP authors. The MW+fiber hybrid network evolves with budget. <https://tinyurl.com/3vakxccm>, 2021. [Online; accessed 11-March-2021].
- [27] Mark Claypool, David LaPoint, and Josh Winslow. Network analysis of Counter-Strike and Starcraft. In *IEEE Performance, Computing, and Communications Conference*, 2003.
- [28] Federal Communications Commission. Universal Licensing System. <http://wireless2.fcc.gov/UlsApp/UlsSearch/searchLicense.jsp>. [Online; accessed 11-March-2021].
- [29] CommScope. HSX8-107-D3A. <https://objects.eanixter.com/PD354739.PDF>, 2012. [Online; accessed 28-July-2021].
- [30] Copernicus by ECMWF. ERA5 hourly data on single levels from 1979 to present. <https://cds.climate.copernicus.eu/cdsapp#!/dataset/reanalysis-era5-single-levels?tab=overview>, 2018. [Online; accessed 11-March-2021].
- [31] DARPA. Novel Hollow-Core Optical Fiber to Enable High-Power Military Sensors. <http://www.darpa.mil/news-events/2013-07-17>, 2013. [Online; accessed 11-March-2021].
- [32] Digital Commerce 360. US ecommerce grows 44.0% in 2020. <https://www.digitalcommerce360.com/article/us-ecommerce-sales/>, 2021. [Online; accessed 28-July-2021].
- [33] DragonWave-X. Services & Support / Pre Deployment / Line of Sight. <https://www.dragonwave-x.com/services/pre-deployment/line-sight>, 2021. [Online; accessed 11-March-2021].
- [34] Ramakrishnan Durairajan, Paul Barford, Joel Sommers, and Walter Willinger. InterTubes: A study of the US long-haul fiber-optic infrastructure. In *ACM SIGCOMM*, 2015.
- [35] Joan Engebretson. Broadband Data Usage Report: Internet-only Homes Use Almost Twice as Much Data as Bundled Homes. <https://www.telecompetitor.com/broadband-data-usage-report-internet-only-homes-use-almost-twice-as-much-data-as-bundled-homes/>, 2019. [Online; accessed 11-March-2021].
- [36] Facebook connectivity. Magma. <https://connectivity.fb.com/magma/>, 2021. [Online; accessed 11-March-2021].
- [37] Facebook connectivity. Rural Access. <https://connectivity.fb.com/rural-access/>, 2021. [Online; accessed 11-March-2021].
- [38] Facebook connectivity. Terragraph. <https://connectivity.fb.com/terraphgraph/>, 2021. [Online; accessed 11-March-2021].
- [39] Federal Communications Commission. Antenna Structure Registration Database. <https://www.fcc.gov/antenna-structure-registration>, 2018. [Online; accessed 11-March-2021].
- [40] Riot Games. Fixing the Internet for real-time applications. <https://goo.gl/SEoxW2>, 2016. [Online; accessed 11-March-2021].
- [41] M. R. Garey and D. S. Johnson. The rectilinear Steiner tree problem is NP-complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977.
- [42] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2016.
- [43] Nikola Gvozdiev, Stefano Vissicchio, Brad Karp, and Mark Handley. Low-latency routing on mesh-like backbones. *ACM HotNets*, 2017.
- [44] Mark Handley. Delay is not an option: Low latency routing in space. In *ACM HotNets*, 2018.
- [45] Jonas Hansryd and Jonas Edstam. Microwave capacity evolution. *Ericsson review*, 1:22–27, 2011.
- [46] Devindra Hardawar. Samsung proves why 5G is necessary with a robot arm. <https://goo.gl/3gZTn8>, 2016. [Online; accessed 11-March-2021].
- [47] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, 2013.
- [48] ITU. Specific attenuation model for rain for use in prediction methods. http://www.itu.int/dms_pubrec/itu-r/rec/p/R-REC-P.838-3-200503-I!!PDF-E.pdf, 2005. [Online; accessed 11-March-2021].
- [49] Ixia. Measuring Latency in Equity Transactions. http://ixia.cabanday.com/products/_content/wp-measuring-latency.pdf, 2012. [Online; accessed 11-March-2021].
- [50] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hözle, Stephen Stuart, and Amin Vahdat. B4: experience with a globally-deployed software defined wan. In *ACM SIGCOMM*, 2013.

- [51] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *ACM SIGCOMM*, 2005.
- [52] Simon Kassing, Debopam Bhattacherjee, André Baptista Águas, Jens Eirik Saethre, and Ankit Singla. Exploring the “Internet from space” with Hypatia. In *ACM IMC*, 2020.
- [53] Kevin Kelly. How much does one search cost? <http://kk.org/thetechnium/how-much-does-a-search-cost>, 2007. [Online; accessed 11-March-2021].
- [54] Kuiper Systems LLC. Application of Kuiper Systems LLC for Authority to Launch and Operate a Non-Geostationary Satellite Orbit System in Ka-band Frequencies. https://licensing.fcc.gov/myibfs/download.do?attachment_key=1773885, 2019.
- [55] Gregory Laughlin, Anthony Aguirre, and Joseph Grundfest. Information transmission between financial markets in Chicago and New York. *Financial Review*, 2014.
- [56] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *ACM MobiSys*, 2015.
- [57] Greg Linden. Make Data Useful. <https://slideplayer.com/slide/4203392/>, 2006. [Online; accessed 11-March-2021].
- [58] McKay Brothers LLC. Quincy Extreme Data Latencies. <http://www.quincy-data.com/product-page/#latencies>, 2017. [Online; accessed 11-March-2021].
- [59] Brian Louis. Trading Fortunes Depend on a Mysterious Antenna in an Empty Field. <https://goo.gl/82kzXd>, 2017. [Online; accessed 11-March-2021].
- [60] Donald MacKenzie. *Trading at the Speed of Light: How Ultrafast Algorithms Are Transforming Financial Markets*. Princeton University Press, 2021.
- [61] Macrotrends LLC. Amazon Net Profit Margin 2006-2021. <https://www.macrotrends.net/stocks/charts/AMZN/amazon/net-profit-margin>, 2021. [Online; accessed 28-July-2021].
- [62] Trevor Manning. *Microwave Radio Transmission Design Guide*. Artech House, 2009.
- [63] Ginny Marvin. Report: Google earns 78% of \$36.7B US search ad revenues, soon to be 80%. <https://goo.gl/kp4L5X>, 2017. [Online; accessed 11-March-2021].
- [64] Microsoft Azure. Content Delivery Network pricing. <https://azure.microsoft.com/en-us/pricing/details/cdn/>, 2018. [Online; accessed 11-March-2021].
- [65] NASA. Precipitation Processing System Data Ordering Interface for TRMM and GPM (STORM). <https://storm.pps.eosdis.nasa.gov/storm/>, 2015. [Online; accessed 11-March-2021].
- [66] NASA Jet Propulsion Laboratory. U.S. Releases Enhanced Shuttle Land Elevation Data. <https://www2.jpl.nasa.gov/srtm/>, 2015. [Online; accessed 11-March-2021].
- [67] NEC. SEA-US: Global Consortium to Build Cable System Connecting Indonesia, the Philippines, and the United States. <https://tinyurl.com/ybj9nhp3>, August 2014. [Online; accessed 11-March-2021].
- [68] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for http. In *USENIX ATC*, 2015.
- [69] A. Nordrum. Fiber optics for the far north [news]. *IEEE Spectrum*, 52(1):11–13, January 2015.
- [70] ns-3 community. Network simulator ns-3. <https://www.nsnam.org>, 2011. [Online; accessed 11-March-2021].
- [71] Jan Odvarko. Har 1.2 spec. <http://www.softwareishard.com/blog/har-12-spec>, 2007. [Online; accessed 11-March-2021].
- [72] Pingzapper. Pingzapper Pricing. <https://pingzapper.com/plans>, 2018. [Online; accessed 11-March-2021].
- [73] Enric Pujol, Philipp Richter, Balakrishnan Chandrasekaran, Georgios Smaragdakis, Anja Feldmann, Bruce M. Maggs, and Keung-Chi Ng. Back-office web traffic on the Internet. In *ACM IMC*, 2014.
- [74] radiowaves. SHPD8-1011. <https://www.radiowaves.com/getmedia/b1a7277f-fde0-4c05-a5fc-7c22c29c5b3a/HPD8-1011.aspx>, 2018. [Online; accessed 28-July-2021].
- [75] radiowaves. SPD8-11. <https://www.radiowaves.com/getmedia/f942ec58-9999-4607-a165-fd4db4def60/SPD8-11.aspx>, 2018. [Online; accessed 28-July-2021].
- [76] Eduard Sackinger. *Analysis and Design of Transimpedance Amplifiers for Optical Receivers*. John Wiley & Sons, 2017.
- [77] William Sentosa, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Haitham Hassanieh, Bruce Maggs, and Ankit Singla. Accelerating mobile applications with parallel high-bandwidth and low-latency channels. In *ACM HotMobile*, 2021.

- [78] Michael Sheetz. SpaceX prices Starlink satellite internet service at \$99 per month, according to e-mail. <https://www.cnbc.com/2020/10/27/spacex-starlink-service-priced-at-99-a-month-public-beta-test-begins.html>, 2020. [Online; accessed 11-March-2021].
- [79] Shkilko, A. and Sokolov, K. Every Cloud Has a Silver Lining: Fast Trading, Microwave Connectivity and Trading Costs. <https://ssrn.com/abstract=2848562>, 2016. [Online; accessed 11-March-2021].
- [80] SimilarWeb. Overview: amazon.com. <https://www.similarweb.com/website/amazon.com/#overview>, 2021. [Online; accessed 28-July-2021].
- [81] SpaceX Starlink. <https://www.spacex.com/webcast>, 2017. [Online; accessed 11-March-2021].
- [82] Starlink Services. Petition of Starlink Services, LLC for designation as an eligible telecommunications carrier. <https://ecfsapi.fcc.gov/file/1020316268311/Starlink%20Services%20LLC%20Application%20for%20ETC%20Designation.pdf>, 2021. [Online; accessed 11-March-2021].
- [83] statista. Online gaming - statistics & facts. <https://www.statista.com/topics/1551/online-gaming/>, 2021. [Online; accessed 28-July-2021].
- [84] Internet Live Stats. Google Search Statistics. <https://www.internetlivestats.com/google-search-statistics/>. [Online; accessed 11-March-2021].
- [85] Steam. Steam & game stats, 2017. <http://store.steampowered.com/stats/> [Online; accessed 11-March-2021].
- [86] Telesat. Telesat: Global Satellite Operators. <https://www.telesat.com/>, 2020. [Online; accessed 11-March-2021].
- [87] Unwired Labs. OpenCellID Tower Database. <https://opencellid.org/>, 2018. [Online; accessed 11-March-2021].
- [88] USGS. National Elevation Dataset (NED). <https://www.usgs.gov/core-science-systems/national-geospatial-program/national-map>. [Online; accessed 11-March-2021].
- [89] J. H. Winters, J. Salz, and R. D. Gitlin. The impact of antenna diversity on the capacity of wireless communication systems. *IEEE Transactions on Communications*, 42(2/3/4):1740–1751, Feb/Mar/Apr 1994.
- [90] X, the moonshot factory. Taara – Expanding global access to fast, affordable internet with beams of light. <https://x.company/projects/taara/>, 2018. [Online; accessed 11-March-2021].
- [91] Xiufeng Xie, Xinyu Zhang, and Shilin Zhu. Accelerating mobile web loading using cellular link information. In *ACM MobiSys*, 2017.

A TOPOLOGY DESIGN

Picking a subset of site-to-site links to connect a set of cities involves solving a typical network design problem. The Steiner-tree problem [41] can be easily reduced to this problem, thereby establishing hardness. Standard approximation algorithms, like linear program relaxation and rounding, yield sub-optimal solutions, which although provably within constant factors of optimal, are insufficient in practice. We develop a simple heuristic, which, by exploiting features specific to our problem setting, obtains nearly optimal solutions.

Inputs: Our network design algorithm requires:

- A set of sites to be interconnected, v_1, v_2, \dots, v_n .
- A traffic matrix H specifying the relative traffic volume $h_{ij} \in [0, 1]$ between each pair v_i and v_j .
- The geodesic distance d_{ij} between each v_i and v_j .
- The distance along the shortest, direct MW path between each pair, m_{ij} , as well as its cost, c_{ij} . This is part of the output of step 1.
- The optical fiber distance between each pair, o_{ij} , which we multiply by 1.5 to account for fiber's higher latency.
- A total budget B limiting the maximum number of bidirectional MW links that can be built.

Expected output: The algorithm must decide which direct MW links to pick, i.e., assign values to the corresponding binary decision variables, x_{ij} , such that the total cost of the picked links fits the budget, i.e., $\sum_{ij} x_{ij} c_{ij} \leq B$. Our objective is to minimize, per unit traffic, the mean stretch, i.e., the ratio of latency to c -latency, where c -latency is the speed-of-light travel time between the source and destination of the traffic.

Problem formulation: Expressing such problems in an optimization framework is non-trivial: we need to express our objective in terms of shortest paths in a graph that will itself be the *result*. We use a formulation based on network flows.

Each pair of sites (v_s, v_t) exchanges h_{st} units of flow. To represent flow routing, for each potential link ℓ , we introduce a binary variable $f_{stij,m}$ which is 1 iff the $v_s \rightarrow v_t$ flow is carried over the microwave link $v_i \rightarrow v_j$, and a binary variable $f_{stij,o}$ which is 1 iff the same flow is carried over the optical link⁸ $v_i \rightarrow v_j$. The objective function is:

$$\min \sum_{s,t} \frac{h_{st}}{d_{st}} \sum_{i,j} (o_{ij} f_{stij,o} + m_{ij} f_{stij,m}) \quad (3)$$

The h_{st} term achieves our goal of optimizing *per unit traffic*. The $\frac{1}{d_{st}}$ term achieves our goal of optimizing the *stretch*.

⁸A “link” between sites can use multiple physical layer hops, both for MW and fiber. The underlying multi-physical-hop distances are already captured by the inputs o_{ij} and m_{ij} so the optimization views it as a single link.

For brevity, we omit the constraints, which include: flow input and output at sources and sinks; flow conservation; total budget; and the requirement that only links that are built ($x_{ij} = 1$) may carry flow. All variables are binary, so flows are “unsplittable” (carried along a single path) and the overall problem is an integer linear program (ILP).

Note that we have decomposed the problem so that link capacity is *not* a constraint in this formulation: MW links will be built with sufficient capacity in step 3; fiber links are assumed to have plentiful bandwidth at negligible cost relative to MW costs. As a result, the objective function will guide the optimizer to direct each $v_i \rightarrow v_j$ flow along the shortest path of built links, which is the direct MW link $v_i \rightarrow v_j$ if it happens to be built, or otherwise, a path across some mix of one or more fiber and MW links.

ILP’s limited scalability: The exact ILP is not scalable, which is the reason we use multiple heuristics, as discussed in §3. As we show in Fig. 8a, the exact ILP, without using our observations on the problem structure, is too computationally inefficient to scale to this scenario. We use subsets of all 120 cities to assess scalability, with the budget proportional to the number of cities in each test, with a budget of 6,000 towers at the largest scale. Even after 2 days of compute, the exact ILP was unable to obtain a result for sets of cities larger than 50. In contrast, our cISP design heuristic is able to solve the problem at the full scale. Second, as Fig. 8b shows, at small scales, where we can also run the exact ILP, our heuristic yields the optimal result. We also tested a linear program rounding approach, but even the naive LP relaxation followed by rounding did not scale beyond 60 cities, and gave results worse than optimal.

B ROUTING & QUEUING

The HFT industry’s point-to-point MW deployments demonstrate end-to-end application layer latencies within 1% of c -latency, after accounting for all delays in microwave radios, interfacing with switching equipment and servers, and application stacks. Such low latencies across point-to-point long-distance links place sharp focus on any latencies introduced at routers for switching, queuing, and transmission.

Internet routers can forward packets in a few tens of microseconds, and specialized hardware can hit 100× smaller latencies [49]. Transmitting 1500 B frames at 1 Gbps takes 12 μ s. Thus forwarding and transmission even across many long-distance links incur negligible latency. Longer routes and queuing delays, however, can have substantial impact.

To assess the impact of routing and queuing in cISP, we use ns-3 [70]. We use UDP traffic with a uniform packet size of 500 bytes. We use the built-in FlowMonitor [19] to measure delay and loss rate, and add a new monitoring module to track link-level utilization. All experiments simulate 100 Gbps of network traffic for one second of simulated time. An experiment takes approximately 10 hours to complete on a single core of a 3.1 GHz processor. Even achieving this running time

requires some compromises: we aggregate the bandwidth of parallel links and remove the individual tower hops to focus on network links between the routing sites.

Routing schemes: Besides ns-3’s default shortest path routing, we implement two other schemes – throughput optimal routing, and routing that minimizes the maximum link utilization, a scheme commonly employed by ISPs [51].

Results: When the traffic and routing match the design target, i.e., the population-product traffic routed over shortest paths, we find that the network can be driven to high utilization (95%) with near-zero queuing and loss. Non-shortest-path routing schemes needlessly compromise on latency in such scenarios. (Plots for this easy scenario are omitted.)

We also test the network’s behavior under deviations from the designed-for traffic model. We emulate scenarios where a city produces more or less traffic than expected by allowing, for each city, a “population perturbation” — each city’s population is re-weighted by a factor drawn from the uniform distribution $U[1 - \gamma, 1 + \gamma]$ for a chosen $\gamma \in [0, 1]$.

Fig. 9a and Fig. 9b show the results for $\gamma \in \{0.1, 0.3, 0.5\}$. Even for large perturbations, the mean delay does not increase by more than 0.1 ms and the loss rate is zero up to an aggregate load of 70% of the capacity designed for, even with just shortest path routing. Other routing schemes are indeed more resilient to higher load, achieving virtually zero loss and queuing delay even at high utilization, but at the cost of latency. For the tested topology, both the alternative routing schemes incur 10% higher latency on average (not shown in the plots). These results indicate there would be significant value in work that reduces the amount of over-provisioning required by making modest compromises on latency on some routes, e.g., as in [43].

Speed mismatch: The bandwidth disparity between the network core and edge for cISP may seem atypical, in the sense that in most settings, the core has higher bandwidth links compared to the edge, while in cISP, edge links (such as those at large data center end points) may often have much higher line rates when they feed their outgoing traffic into cISP. Thus, we also evaluate if this “speed mismatch” causes persistent congestion at cISP’s ingresses.

We run ns-3 simulations with several sources (S_i) connected to a sink (D) through the same intermediate node (M). The $M-D$ link rate is fixed at 100 Mbps. We then evaluate settings with every S_i-M link being either 100 Mbps or 10 Gbps. The former is the control, and the latter is the setting with a speed mismatch. M has an unbounded queue. Ten sources send 100 KB TCP flows (small, as is expected in cISP) to the sink, D . The arrival of these TCP flows follows a Poisson process, consuming on average 70% of the $I-D$ link’s bandwidth. Each simulation run lasts 10 s and we conduct 100 such runs. We test TCP both with and without pacing.

Fig. 10a shows that the median queue occupancy at M is higher without pacing, especially at the 95th percentile. However with pacing, queueing behavior is nearly the same.

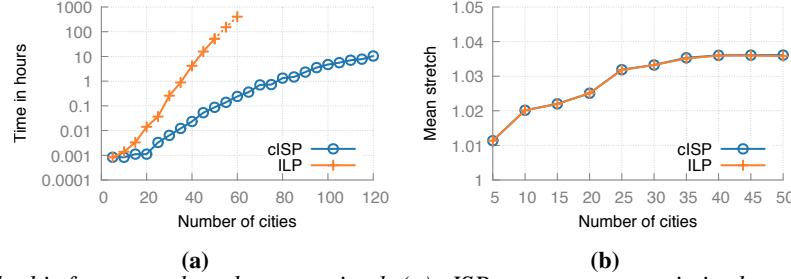


Fig. 8: cISP’s design method is fast-enough and near-optimal: (a) cISP generates an optimized topology within hours for 120 cities while the ILP does not yield a result even after 2 days for more than 50 cities. For the ILP, runtimes for 50+ cities are extrapolated by curve fitting. (b) The stretch achieved by cISP matches that of the ILP to two decimal places for instances that can be optimized by the ILP.

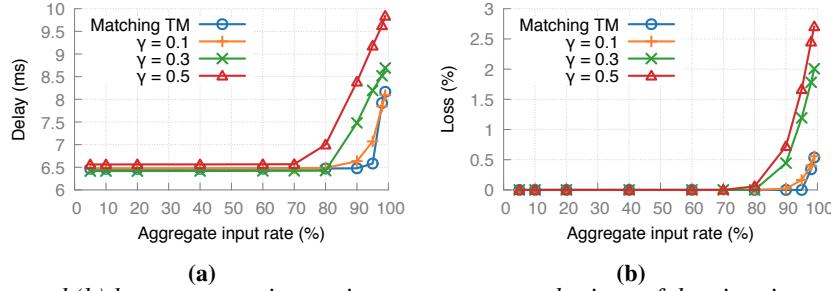


Fig. 9: (a) Average delay and (b) loss rate remain consistent across perturbations of the city-city traffic model, except under heavy load.

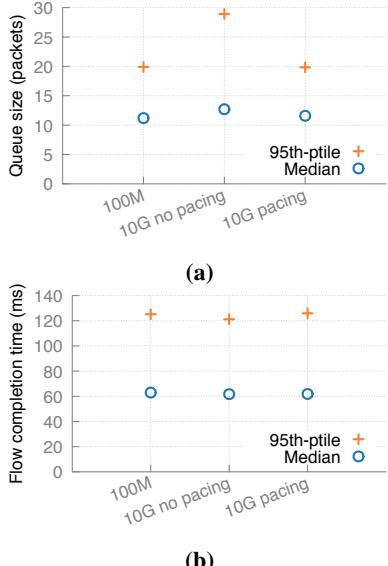


Fig. 10: TCP pacing addresses the problem of capacity mismatch (a) by reducing persistent queuing (b) without affecting flow completion times.

The median flow completion times (Fig. 10b) are unaffected both with and without pacing.

C FURTHER DESIGN CONSIDERATIONS

C.1 Is the city-city traffic model special?

Ideally, we would be able to use wide-area traffic matrices from some ISP or content provider for modeling. In the absence of such data, we focus on showing that cISP can be tailored to vastly different deployment scenarios and their cor-

responding traffic models. Apart from the city-city population product model, we use (a) traffic between a provider’s data centers; and (b) traffic between the cities and data centers.

An inter data center cISP: We use Google data centers as an example, considering all 6 publicly available US locations - Berkeley, SC; Council Bluffs, IA; Douglas County, GA; Lenoir, NC; Mayes County, OK; and The Dalles, OR. In the absence of known inter-data center traffic characteristics, we provision equal capacity between each DC-pair.

Data centers to the edge: We also model a scenario where data centers are to be connected to edge locations in cities. Each of the 120 cities connects to its closest Google data center, with traffic proportional to its population.

We show in Fig. 11 that using the same design approach as in §3, both of the above scenarios result in networks with lower cost than the city-city model. Thus, cISP can be tailored to a variety of use cases and traffic models.

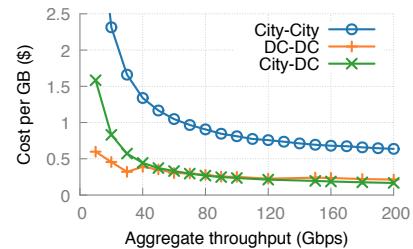


Fig. 11: Cost per GB for different traffic models: the City-City model, discussed in the most detail, is the most expensive.

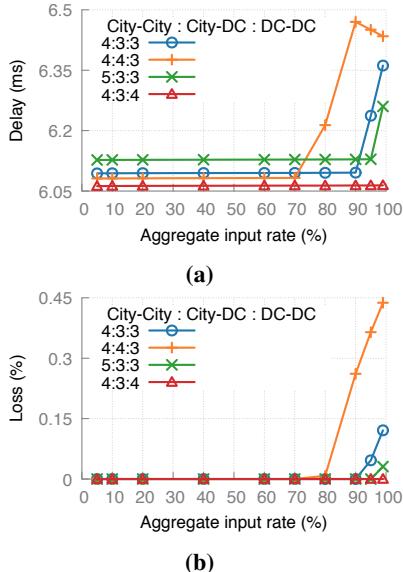


Fig. 12: (a) Average delay and (b) loss rate remain consistent across deviations from the designed-for traffic mix, except under heavy load.

C.2 Traffic model mismatches

A cISP may carry a mix of city-city, inter-DC, and DC-edge traffic. How does its performance degrade as the *proportion* of these traffic types departs from the design assumptions?

We design a cISP to carry an aggregate of 100 Gbps with a city-city : DC-edge : inter-DC traffic proportion of 4:3:3. Using ns-3 simulations similar to those in §B, we then test this network under several traffic mixes different from this designed-for mix — 5:3:3, 4:3:4, and 4:4:3.

Fig. 12a and Fig. 12b show that there is a difference of less than 0.05ms in mean delay across different combinations of traffic matrices up to an aggregate load of 70% of the design capacity. Similarly, loss remains nearly 0 until this load. The decrease in delay at high load (4:4:3 for $x > 90$ in Fig. 12b) is due to losses, which are likelier on longer, higher-delay paths.

Mean delay depends more on city-city traffic, as expected: city-city traffic requires a wider infrastructure footprint, and deviations from its design parameters have greater impact.

Thus, as discussed in §B, significant traffic model deviations can be absorbed using some over-provisioning, in line with current ISP practices.

C.3 Is the US geography special?

It is reasonable to ask: are the population distribution and geography of the U.S. especially amenable to this approach, or is it applicable more broadly? The availability of high-quality tower data and geographical information systems data for the U.S. enables a thorough analysis. While similar data is, unfortunately, not available to us for other geographies, we can approximately assess the design of a cISP in Europe using public, crowd-sourced data on cellular towers [87]. Lacking fiber conduit data, we assume that fiber distances between

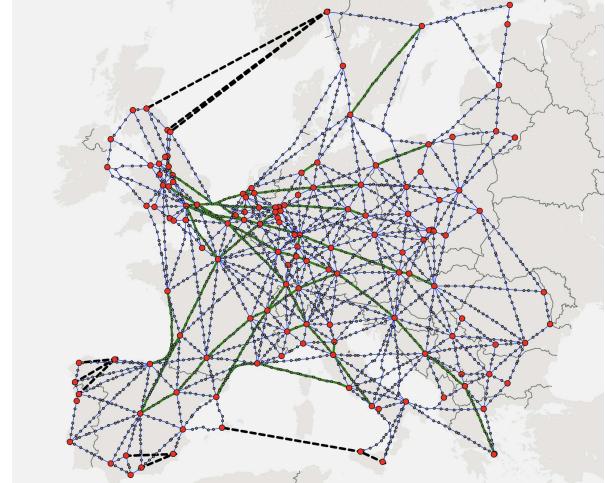


Fig. 13: A 100 Gbps 1.04 \times stretch cISP across Europe. This network uses several fiber connections (dashed, black lines).

cities are inflated over geodesic distance in the same way as in the US ($\sim 1.9 \times$). Using our methodology in §3, we design a European cISP of similar geographical scale across cities with population more than 300k, targeting the same aggregate capacity and mean latency (1.04 \times here vs. 1.05 \times for cISP-US). The cost of this design, shown in Fig. 13, is similar as well, with $\sim 3k$ towers. Note that the impact of Europe’s higher population density is not seen here, because we explicitly design for the same aggregate throughput. One could, alternatively, normalize throughput per capita, and compare cost per capita, to obtain similar results.

Admittedly, there is not yet a known approach to bridging large transoceanic distances using MW, limiting our approach to large contiguous land masses that need to be interconnected with fiber. In the distant future, LEO satellite links, hollow-core fiber, or even towers on floating platforms may be of use for such connectivity.

D AMERICAN TOWER DEPLOYMENT

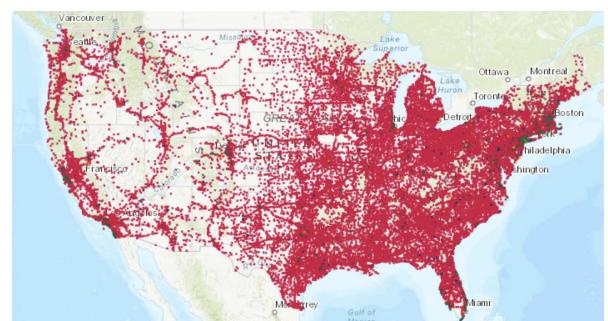


Fig. 14: American tower deployment as per 5th March, 2021.

American Tower [7] claims to have a presence at more than 42,000 tower sites across the US, as of 5th March 2021. Fig. 14 shows their current deployment. We could not access their database due to legal bindings.

Configanator: A Data-driven Approach to Improving CDN Performance.

Usama Naseer
Brown University

Theophilus A. Benson
Brown University

Abstract

The web serving protocol stack is constantly evolving to tackle the technological shifts in networking infrastructure and website complexity. As a result of this evolution, web servers can use a plethora of protocols and configuration parameters to address a variety of realistic network conditions. Yet, today, despite the significant diversity in end-user networks and devices, most content providers have adopted a “one-size-fits-all” approach to configuring the networking stack of their user-facing web servers (or at best employ moderate tuning).

In this paper, we demonstrate that the status quo results in sub-optimal performance and argue for a novel framework that extends existing CDN architectures to provide programmatic control over a web server’s configuration parameters. We designed a data-driven framework, Configanator, that leverages data across connections to identify their network and device characteristics, and learn the optimal configuration parameters to improve end-user performance. We evaluate Configanator on five traces, including one from a global content provider, and evaluate the performance improvements for real users through two live deployments. Our results show that Configanator improves tail (p95) web performance by 32-67% across diverse websites and networks.

1 Introduction

Web page performance significantly impacts the revenue of content distribution networks (CDNs) (e.g., Facebook, Akamai, or Google), with studies showing that a 100ms decrease in page load times (PLT) can lead to 8% better conversion rate for retail sites [14, 30]. Yet, uniformly improving web performance is becoming increasingly challenging due to the growing disparity in the network conditions (e.g. bandwidth, RTT) [3, 36, 120, 135] and end-user devices [93, 94, 108, 134]. To address this disparity and improve the quality of experience (QoE), the networking community is constantly developing new protocols and configuration parameters for web servers (AKA, edge servers), e.g., PCC [31], BBR [23], QUIC [51], etc.

The optimal choice of configurations is contingent on

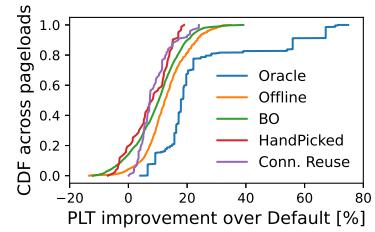


Figure 1: Comparison of various tuning techniques.

the network infrastructure [3, 36, 77, 98, 120, 135, 142], website complexity [20, 21, 95, 134, 136], and end-user devices [1, 94, 108]. Furthermore, innovations along any one of these dimensions will lead to changes to default parameters and new protocols. Although different regions and ISPs have radically different networking infrastructure and mobile devices [1, 93], a majority of CDNs continue to employ a “one-size-fits-all” [49] approach to configuring their edge servers, which results in sub-optimal performance [3, 36, 135] and high tail-latency in certain regions [142].

1.1 Configuration Tuning Status-Quo

Most attempts to tackle this growing diversity involve manually analyzing the performance of configuration options across different regions [49], devices [1], or websites [110, 135]. While several CDNs expose configuration knobs to their customers [40, 45], it is challenging to take the full advantage of the knobs due to the required manual efforts and the lack of automated learning techniques for effective tuning.

This paper focuses on tuning a broad set of configuration knobs across the transport (e.g., congestion control algorithm) and application layers (e.g., HTTP version) as highlighted in Table 1. Next, we illustrate the challenges and benefits of dynamically tuning network configurations.

Challenges in tuning stack: In Figure 1, we illustrate the difficulty of tuning configurations by comparing page load time (PLT) of popular websites, when configured using popular tuning techniques (setup explained in § 2.2). Specifically, *Bayesian Optimization* [104] (e.g., CherryPick [4]) a statistical

technique used for tuning systems configurations [4, 32, 75, 131], operator *hand-tuned* configurations (discussed in § 2), *TCP connection reuse* a traditional optimization (discussed in § 6.5), and a closed-loop *offline-learning* technique. We compare their performance against two baselines: *optimal* configuration discovered through an exhaustive brute-force search, and *default* configurations for Linux and Apache (Table 1).

Hand-tuned configurations are manually selected and are thus, coarse-grained. While they out-perform the *default* at median, they fail to provide optimal performance across varying network conditions and may even lead to performance degradation for some networks. TCP connection reuse only optimizes a subset of knobs (e.g., initial congestion window) and is unable to take full advantage of the diverse network stack knobs. Bayesian optimization aims to quickly discover “good” configuration. While fine-grained, this approach is relatively static and does not re-evaluate old choices, and is thus unable to adapt to network dynamics [78]. We observe the effects of this rigid behavior with wildly varying tail performance. Lastly, we explore an offline model which learns on traces from prior days and applies the learned model on connections for the next day. Offline modeling is fine-grained but with limited dynamicity: the trained model is unable to react to real-time issues. Unfortunately, due to the high dimensionality of the Internet’s dynamics, these real-time issues are the norm, not the exception [67, 69, 78]. We observe in Figure 1 that offline performs closest to the optimal but still falls short because of its inability to react in real-time.

Our brief analysis of tuning approaches highlights the need for a dynamic, fine-grained approach to tuning configurations.

1.2 Configanator

In this paper, we eschew the notion of a homogeneous approach to tuning web server configurations and instead argue for a “curated” approach for configuring on a per-connection basis. In particular, we argue that edge servers should be configured to serve each of the incoming connections with the optimal protocols and configuration parameters, e.g., a web server may employ Cubic in favor of BBR when serving a low bottleneck buffer connection [111, 114]. To this end, we argue for a simple but robust server architecture that introduces flexibility into the network stack, enables reconfiguration, and systematically controls configuration heterogeneity. We also introduce a contextual multi-armed bandit based learning algorithm, an embodiment of domain-specific insights, which tunes configuration in a principled manner to find optimal configurations in minimal time. Taken together the design and the learning algorithm, our system, *Configanator*, enables a CDN to systematically explore heterogeneity in a dynamic and fine-grained manner while improving end-user performance. The design of Configanator faces several practical challenges:

- Network dynamics: network may change every few minutes [67, 83, 146] and thus requires continuous learning.

Layer	Protocol Options	Default	Example parameter
Transport	congestion_control (CC)	Cubic	BBR, Cubic, Reno
	initial congestion window	10 MSS	Integer (1, 4, 30)
	slow_start_after_idle	1	boolean {true, false}
	low_latency	0	boolean {true, false}
	autocorking	1	boolean {true, false}
	initRTO	1s	decimal (0,3,1)s [59]
	pacing (fair-queue)	0	boolean {true, false}
	timestamps	1	boolean {true, false}
Web App	wmem	{4096}B	{163840}B
	HTTP Protocol	1,1	1,1, 2
	H2 push	On	On, Off
	H2 max header list size	16384B	Integer values
	H2 header table size	4096B	Integer values
	H2 max concurrent streams	100	Integer values
	H2 initial window size	65535B	Integer values < 2 ³¹
	H2 max frame size	16384B	Integer values < 2 ²⁴

Table 1: Web stack configuration parameters.

- Non-Gaussian noise: CDNs focus on improving tail latency [27, 53, 145] which is often caused by non-Gaussian processes (e.g., last-mile contention [125], mobile device limitations) and are difficult to model.
- High-Dimensionality: Content personalization, diverse devices [94, 134], and last-mile connections [125] introduce high dimensionality that limits the efficacy of offline closed-loop approaches [67, 68].
- High data cost: Generating data for learning requires testing configurations and may disrupt user’s performance. Hence, the negative impact on users must be minimized.
- Limited flexibility: Linux kernel and modern web servers lack the flexibility to tune configurations on a per-connection basis, thus requires enhancing the traditional networking stack.

The key insight of Configanator is to simultaneously operate in two modes depending on the “quality” of the performance model. Essentially, Configanator intelligently selects samples that speed up model convergence, then at steady-state it transitions into a greedy-mode that stochastically samples points to iteratively improve performance. Configanator further clusters similar connections together and samples across clusters to amortize the cost of exploration.

Configanator uses a contextual multi-armed bandit [133] designed explicitly to continuously converge to an optimal (or near-optimal) configuration within a minimal number of exploration steps. Our ensemble fuses the stateful exploration of Gaussian-bandit with the non-determinism of Epsilon-bandit, enabling informed exploration of the configuration space while randomly re-sampling old configurations. The re-evaluation of data samples enables Configanator to directly tackle non-Gaussian noise within the domain. The data collected by the ensemble is encoded in a decision tree – which enables quick and easy classification but is also amenable to automatic generation of rules for a CDN’s web server.

To demonstrate the benefits, we conducted large-scale simulations and live deployments. We used datasets from a *GlobalCDN* and public datasets from CAIDA [22], MAWI [8], Pantheon [142] and FCC [41]. Our simulation results show that Configanator provides 32-67% (up to 1500ms) improvement in the PLT at tail (p95) across the different traces. Given the

Layer	Option	Top configs. in N.A. (cross-CDN)	% of CDNs configuring differently across regions	Example of observed cross-regional difference
Web App	HTTP version	H1.1(44.3%), H2(55.7%)	4.7%	N.A. H2 -> Asia H1.1
	Max header list size	16384 (100%)	0%	None
	Header table size	4096 (100%)	0%	None
	Max concurrent streams	100 (44%), 128 (56%)	1%	N.A. 100 -> EU 128
	Initial window size	65536 (71%), 65535 (15%), >1M (14%)	1.9%	N.A. 1048576B -> Asia 65535B
Transport	Max frame size	16,777,215 (81%), 16384 (19%)	0%	None
	ICW	{10 (62%), 4 (20.5%), 24 (5.3%)} MSS	6.9%	N.A. 24 MSS -> Asia 10 MSS
	initRTO	{0.3 (9.2%), 1 (82.6%), 3 (8.2%)} sec	2.3%	N.A. 3s -> EU 1s
	RWIN	{29200 (57.4%), 14600 (8.2%), 42780 (6.8%)} bytes	3.6%	N.A. 29200B -> Asia 12960B

Table 2: Heterogeneity in configs. across 5 regions

recent arms race by CDNs to improve web performance, we believe that Configanator’s modest improvements will result in significant revenue savings [14, 19, 30, 103]. Please refer to the project website ¹ for the related resources.

2 Empirical Study

Next, we analyze CDNs to determine the current extent of configuration tuning (§ 2.1) and quantify its implications (§ 2.2).

2.1 Fingerprinting web configurations

We aim to understand if modern CDNs employ homogeneous configurations, as suggested by anecdotal evidence, or heterogeneous configurations to tackle diversity in the Internet’s ecosystem. To this end, we developed a tool to infer and fingerprint a web server’s [49, 92] application/L7 and transport/L4 layers configuration parameters by actively probing the servers and inspecting the packet headers and their reaction to emulated network events (e.g., packet loss). Please refer to Appendix A for more details about the tool. Using the tool, we fingerprinted the configurations for the Alexa top 1k websites from five different regions (North America (N.A), South America, Asia, Europe, and Australia), and present the results in Table 2. We use N.A configurations as the reference point and compare the observed configurations along two axes:

Observation 1: Heterogeneity across CDNs: In Column 3 (cross-CDN), we observe that different CDNs use different configurations in N.A. While some of the heterogeneity can be attributed to differences in the default values for different OSes, we observe that CDNs do use non-default values, e.g., amazon.com uses an ICW of 24 MSS in N.A.

Observation 2: Homogeneity within a CDN: In Column 4 (cross-region), we observe that only a small number of CDNs tune their network stack to account for regional differences, i.e., use different configurations in N.A. than the other regions. The highest amount of tuning occurs at L4, with 6.9% of the CDNs tuning the ICW differently in N.A. than in other regions, e.g., 24 MSS in N.A. but 10 MSS in Asia for amazon.de.

Takeaway: Taken together, these observations indicate that while individual CDNs perform modest tuning, most do not tune finely enough to account for regional diversity. In fact, only a small set of CDNs configure differently across regions.

2.2 Implications of Configuration Tuning

Next, we quantify the benefits of dynamically tuning a web server’s networking stack by conducting a large scale study in our local testbed. We emulate a wide range of representative networks (extracted from real-world traces [8, 22, 41, 142]) and perform an exhaustive, brute-force search of configuration space (detailed description of the traces is provided in § 6.1). Table 1 lists the set of configurations, with default settings for TCP and HTTP taken from the Linux transport stack (kernel 4.20) and Apache (v2.4.18), respectively. In each trial, the server iteratively selects a configuration from the possible configuration space, a representative network is emulated using NetEM [54], and the PLT of a randomly selected website from Alexa Top-100 (locally cloned on the server) is measured five times. The *optimal* configuration is defined as the one that results in the lowest PLT for a specific network and website.

Figure 1 explores the implications of using sub-optimal configurations, by comparing optimal and default configurations for pageloads across diverse networks and websites. We observe that there is ~18% PLT improvement at the median (over 70% at tail) when optimal configurations are used over the default. While the number may appear small, they can result in tremendous revenue improvements [14, 30], and more in the developing regions where CSPs are investing heavily to improve network [37, 80]. We observe the highest reconfiguration benefits for low bandwidth, high RTT/loss regions, representative of developing region networks.

Next, we analyze congestion control measurements across different regions from Pantheon [142]. We observe that emerging protocols, e.g., BBR, PCC, or Remy, which use probing or ML to improve performance, do not provide uniformly superior performance. In particular, we observed that in many situations BBR is suboptimal, performing 3X to 10X worse than the optimal congestion control. Moreover, no congestion control is optimal for more than 25% of the networks tested, and the median congestion control is optimal for only 6% of the networks.

3 Configanator’s Algorithm

Tuning network configurations to maximize the web performance for diverse networks and end-users presents a complex learning problem. Next, we formulate the problem and present a domain-specific ensemble to address the challenges.

Problem Formulation: Given a set of networking configurations ($C = \{c_1, c_2 \dots c_n\}$), network conditions (N

¹<https://systems.cs.brown.edu/projects/configtron/>

$= \{n_1, n_2 \dots n_n\}$), devices ($D = \{d_1, d_2 \dots d_n\}$), websites ($W = \{w_1, w_2 \dots w_n\}$) and a function, $f()$, that maps a website, network condition, device, and configuration to a metric of web page performance (e.g., PLT or SpeedIndex). Note that, $f(c_i, n_i, d_i, w_i)$ returns the web page performance metric value for applying configuration c_i to a user device d_i loading website w_i in network n_i . In this paper, we use PLT as the metric for web page performance and can be easily replaced with other metrics. Our goal is to solve Eq. 1 and find a configuration (c^*) that minimizes $f()$ for a given combination of n_i , d_i and w_i .

$$\operatorname{argmin}_{c^*} f(c^*, n_i, d_i, w_i) = \{f(c_i, n_i, d_i, w_i) | \forall c_i \in C\} \quad (1)$$

Solving the black-box function $f()$ requires exploring sample space. Two possible exploration algorithms are:

- *Brute force* [2] which tests each possible configuration one by one until the entire space is explored.
- *Bayesian optimization* (BO) [16, 104] is a principled global optimization strategy that uses a prior probability function to capture the relationship between the objective function (Eq 1) and the observed data samples. BO models $f(c, n, d, w)$ as a Gaussian process (GP) [16]. GP is a distribution of candidate objective functions and is used to select the next promising point (c^*) which is then evaluated on a connection. GP then updates its posterior belief by adding the new observation $f(c^*, n, d, w)$ to the set of seen observations. With every new observation, the space of possible candidate functions gets smaller and the prior gets consolidated with the new evidence.

Challenges: Both approaches are sub-optimal for our use-case due to several reasons: (1) non-stationary network conditions [10, 67, 69, 146] (network conditions change every few minutes), (2) BO assumes that data is noise-free or only has Gaussian noise [118], and non-Gaussian noise (tail latency can not be modeled by a Gaussian process [78]) disrupts the estimation of next candidate sample and is observed to impact BO’s hyper-parameters (e.g., threshold on expected improvement for next sample to stop the exploration), (3) costly data collection (collecting data requires testing on end-users which can impacts PLT and revenue), (4) data scarcity (testing on individual users requires each user to generate a tremendous number of connections but a user may only visit the site a few times).

Intuition: The intuition behind Configanator’s algorithm is to decompose the model building into two phases: (i) an initial phase during which the search should be directed to speed up the process and build a good (not perfect) model, and (ii) a steady-state during which the search should be more stochastic to iteratively improve the model and tackle non-Gaussian noise. Building on these insights, Configanator leverages a combination of clustering, an ensemble of bandit-techniques, and ML to address the aforementioned challenges. Specifically, clustering is used to group connections based on their network and device similarity (called *Network Class*) and aggregate observations across similar connections to address data scarcity. The use of a contextual multi-armed bandit [133] enables Configanator to

explore configurations and continuously collect data samples to learn and tackle dynamic client-side conditions in a balanced and online manner. To generalize observations across the connections, a Decision Tree is trained for efficient inference.

3.1 Domain-Specific Multi-Armed Bandit

Configanator’s learning algorithm consists of a contextual multi-armed bandit [76, 84, 133] with three arms:

- **Exploration Arm-1 (Gaussian process [104, 112]):** The Gaussian process (GP) bandit [4, 73] uses an acquisition function to perform a directed search to quickly discover a “good” (might not be optimal) solution when no information exists for a Network Class (NC). There are multiple acquisition functions available [16] and we use *Expected Improvement* (EI) [112] because of its well-documented success [4, 32, 47]. This search process includes two terminating conditions: a threshold on EI and minimum of number of data points to explore. For non-continuous configurations (e.g., HTTP version), we encode them into a number to discretize the space². To account for performance differences between websites and NCs, the GP-arm is composed of a collection of GP models, one for each unique website and NC combination (Appendix D).
- **Exploration Arm-2 (Epsilon-bandit [128]):** The Epsilon-bandit randomly re-samples the data points to overcome issues endemic with the Gaussian process (and Bayesian Optimization in general), e.g., non-stationarity of mean performance. The network operator bounds the random exploration by defining a parameter, ϵ , that controls the trade-off between speed of exploration and the impact on end-user QoE. A high ϵ improves exploration but results in a negative impact on clients’ QoE due to constantly changing configurations.
- **Exploitation Arm (Decision Trees [107]):** The exploitation arm uses ML-powered prediction to model the data collected through the exploration arms. We evaluated several techniques including Support Vector Machines, Decision Trees (D-Trees), and Random Forests. We found that the D-Tree hits the sweet spot, providing comparable accuracy to the other models while being efficient enough to build and update at scale. The D-Tree encompasses all websites and NCs to learn across websites and networks. Leveraging the config-performance curves collected by underlying exploitation arms, a single D-Tree model is trained for the “good” configuration found so far for each website/NC pair, and the D-Tree maps { website, device, network/AS characteristics } to their optimal configuration.

Context-based arm switching: Configanator constantly switches between the arms based on the NC’s “context” which is defined as the quality of the GP-model for the website/NC. It operates in two modes: (i) *Bootstrap*, when no information exists for a website or NC, the context is empty and the GP-arm is used to explore the configuration space in a principled manner until the acquisition function (EI) indicates

²GPyOpt [101] supports mixed (continuous/discrete) domain space [102].

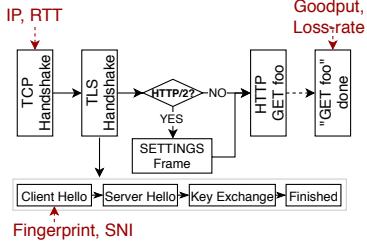


Figure 2: Connection features.

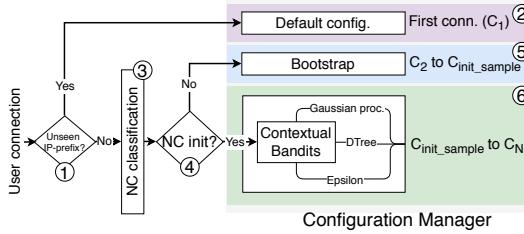


Figure 3: Learning framework workflow.

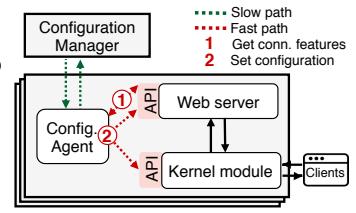


Figure 4: System architecture.

that a good configuration is found, (ii) *steady-state*, when information from the GP-arm indicates “good” configurations, Configanator uses either the epsilon-bandit to further explore the configuration space, or the exploitation arm (i.e., D-Tree) to leverage best configurations. Note that, random exploration through epsilon-bandit continues after EI threshold is met.

3.2 Discovering Network Classes

Configanator extends on observations from prior studies [67, 89] and classifies homogeneous connections into *Network Classes (NC)* with the intuition that similar connection characteristics lead to identical optimal configurations.

Design Goals and NC Features: The ideal NC-clustering should (i) create a small number of clusters, each with a large number of connections to amortize the cost of explorations, and (ii) all members of a cluster should have near-identical profiles. The two goals inherently contradict: the greater the number of entities in an NCs, the higher the probability that the NC contains entities with diverging performance. The second goal is further complicated by the sensitivity of a configuration’s performance (e.g., PLT) to a myriad of factors in the end-to-end connection. To this end, we use network characteristics (bandwidth, latency, loss rate), AS information (ASN, geo-location), and device type as the basis for measuring similarity.

Capturing NC Features: To enable Configanator to effectively tune both the transport and HTTP layers, we must identify all features during the TCP handshake before the HTTP version is negotiated through ALPN [60]. If we identify features after HTTP negotiations, then tuning the HTTP layer would require renegotiation and hence incurs latency penalty. In Figure 2, we highlight the features collected during specific phases of the connection: (1) During the TCP handshake, we capture RTT, IP-prefix, and ASN/geo-location³. (2) During the TLS handshake, we apply TLS fingerprinting techniques [5, 18, 70, 127] on the TLS *Client Hello* to perform device identification and capture device features (accuracy evaluated in Appendix B). Note that, most operators already employ TLS fingerprinting for security purposes [6, 61, 126] and is also supported by major web servers [29]. We use the *Server Name Indication (SNI)* in the *Client Hello* to determine the website hostname which is one of the input features for the

learning framework. (3) For goodput and loss rates, features that cannot be captured during handshake, we build and use a historical archive of these network characteristics.

Network Classification: Clustering can be done using conventional techniques, e.g., K-means, hierarchical, or domain-specific techniques [17, 38, 105], e.g., Hobbit [74] or, CFA [67], or using CDN state of the art [26, 87, 119, 139, 140], e.g., latency-based groups [26, 119, 139]. Although Configanator can incorporate any of the aforementioned techniques, our prototype uses “K-means” clustering because of its simplicity. Configanator empirically selects the smallest K (i.e., the number of classes) that bounds the spread of performance within each NC by a predefined limit⁴ (evaluated in § 6.2 and Appendix D).

3.3 Configanator Workflow

Figure 3 presents the end-to-end workflow. Default configuration is initially used for a newly-seen IP-prefix (1, 2) due to the lack of information about its goodput and loss-rates. For any subsequent connection from the IP-prefix, the recorded, as well as the actively collected features, are used for NC classification (3). If the network, AS and device characteristics do not fit into an existing NC, a new NC is created (4) and the next *init_samples* connections for the respective NC are used for bootstrapping (5) its empty context. When the respective NC is bootstrapped, the multi-armed bandit uses the actively and passively collected features, as well as the requested website, for determining the context and alternates between the arms (6). The connections are correspondingly tuned (7) and the resulting performance metrics are fed back into the models to help refine their classifications and improve accuracy. Due to the computationally intensive nature, Configanator builds/updates NC clusters in the background and uses the already-built clusters for real-time classification.

4 Architecture

Our re-architected web server consists of four components (Figure 4): The *HTTP server application* [39, 97, 121, 132] operates as it does today: serves content and collects performance metrics for each connection. The *Configuration Manager* runs the learning algorithm on the telemetry collected from the web servers. The *Configanator-API* abstracts vendor-specific

³Captured using end-user’s IP and publicly available data (RouteViews for AS [17], MaxMind for geo-location [62])

⁴Controlled by NCSpread knob in simulator (Table 4 in Appendix).

configuration details and provides a uniform interface for configuring web server’s network stack parameters. A *Configuration Agent* runs on each web server and uses the information received from the Configuration Manager to configure the connections through the Configanator-API.

Adopting this architecture in an incrementally deployable manner is practically challenging. The configuration parameters are exposed in an ad-hoc manner, e.g., tuning transport configuration requires *IOCTL* and *setsockopt*, while tuning HTTP requires changes to application code and enhancements to the ALPN protocol. Additionally, most CDNs use well-established code bases and exposing the configuration interfaces required by Configanator should incrementally build on the existing code.

4.1 Configanator-API

The *Configanator-API* presents a uniform interface over the web server’s serving stack thus abstracting away OS and web server specific details. This simplified interface enables the *Configuration Agent* to easily tune the network stack, without having to understand vendor-specific details or implications.

Transport tuning: Unfortunately, the traditional kernels only expose and provide flexible reconfiguration for a subset of TCP’s parameters. In particular, some parameters (e.g., ICW) can be configured on the connection level, while others can only be configured on a global scale (e.g., *tcp_low_latency*). Using Configanator at a coarser granularity, either limits the type of supported connections on a machine or limits the configuration space. There are several options to address this issue ranging from user-space TCP/IP stacks [35, 65, 106], kernel modules, eBPF programs, to leveraging virtualization. We opt for a kernel module-based design over virtualization approaches because hosting a single configuration per VM introduces significant overheads.

HTTP tuning: HTTP version and H2 settings are determined through Application Layer Protocol Negotiation (ALPN) [60] in TLS handshake and H2 SETTINGS [12] frame, respectively. Given the requirement for per-connection tuning, we augment the ALPN and the H2 SETTINGS frame code to enable fine-grained control over these configurations. In particular, Configanator configures these settings by restricting the options presented in the server advertisement to the configuration setting being tuned, e.g., to set the HTTP protocol to H2, we limit the “ALPN next protocol” field in TLS *Server_Hello* to just H2. Similarly, we restrict the options in the SETTINGS frame to configure HTTP/2 settings.

Tuning Workflow: Configanator-API tunes both the TCP and HTTP version during the TLS handshake: after receiving the *Client Hello* from the end-user and prior to sending the *Server Hello*. This is the perfect location to tune because (1) the complete feature set required to determine a connection’s NC and configuration can be captured at this point, and (2) the server is yet to finalize the HTTP protocol, which the ALPN selects in *Server Hello*, thus enabling us to configure

the HTTP version. We note that at this phase of the connection, the TCP state machine is in its infancy because the sender has not sent any data, and thus virtually no significant state is lost when we change the congestion control algorithm or settings.

4.2 Configuration Agent

The *Configuration Agent* is the glue logic between the *Configuration Manager* and *Configanator-API* — it collects the connection features, uses rules provided by the Configuration Manager to make configuration decisions, and configures them using the *Configanator-API*. We select a proactive approach, where the *Configuration Manager* constantly pushes NC and configuration mappings to the *Configuration Agent* which caches them locally. Further for an unseen IP-prefix, *Configuration Agent* uses the default configuration, until the *Configuration Manager* finds a better mapping.

4.3 Configuration Manager

The manager runs in a centralized location, e.g., a data center or locally in a Point of Presence (PoP), with the implications later explored in Appendix F.9. It is charged with running the learning algorithms (§ 3), network classification models (§ 3.2), and disseminating the configuration maps to the *Configuration Agents’ cache*. The *Configuration Manager* disseminates and collects data from the *Configuration Agents* using distributed asynchronous communication. For the NC and configuration maps, *Configuration Manager* broadcasts to all *Configuration Agents*, whereas for reporting performance data and for making one-off-request for configuration maps, the *Configuration Agents* use unicast.

5 Prototype

The implementation highlights of the prototype are as follows: **Configanator-API:** partly resides within the kernel (as a module) and partially resides in user-space in the form of additions to the web server code (in our case Apache). The components within the kernel allow us to tune the transport, while the user-space allows us to tune the HTTP layer.

The kernel module reuses functions provided by kernel’s congestion controls through the *tcp_congestion_ops* interface and tunes fields in appropriate structs (e.g., *inet_connection_sock*). For tuning globally-defined knobs at a per-connection level (e.g., *tcp_low_latency*), we leveraged kernel patches [34, 55] to define and reference them from *tcp_sock* struct. The user-space component within Apache code tunes HTTP version in Apache and its design is generalizable to other servers that use OpenSSL. OpenSSL library is used by most web server implementations and allows web servers to register a *SSL_CTX_set_alpn_select_cb* [44] callback to modify ALPN decisions. To tune HTTP version, we register a call back which looks up the HTTP version to use for a connection and restricts the ALPN options advertised to the one specified by the Configuration Agent. For H2 settings, we modify the Apache H2

module to dynamically select the configurations while sending the SETTINGS frame. The user-space agent also generates the TLS fingerprint for device identification. We use JA3 [70] for TLS fingerprinting. In both our testbed experiments and in the live deployments, we use the ALPN-centric approach which modifies protocol options presented in the advertisements.

Configuration Agent: is user-space code and is implemented in 492 LoC of C++ code. The agent updates TCP and HTTP settings via the Configanator-API. This component also parses Apache’s logs for measuring network characteristics. For measuring the PLT, the web server injects a simple JavaScript into the webpage to measure the navigation timings.

Configuration Manager: is developed in 1435 LoC (Python). It uses SciLearn [116] for D-Tree and GPyOpt [101] for the Gaussian Process. For communication with the Configuration Agents, we use ZeroMQ [144]. For D-Tree, we use SciLearn’s CART algorithm with the following configuration: (i) entropy for the information gain, (ii) set the minimum number of leaf nodes to 80, (iii) set the minimum number of samples needed for the split to 2, and (iv) do not limit the depth of tree. For Gaussian process, we use init_sample=4, min_sample_tested=7 and EI=8% thresholds. We tested a range of these hyper-parameters settings⁵ and selected the ones resulting in the highest accuracy. Following [4], we tested EI threshold in 3-15% range and selected 8% for its best trade-off between accuracy and search cost. For controlling the “K” for NCs, we use a *NCSpread* threshold of 5% (Appendix D).

6 Evaluation

We evaluated Configanator through a large-scale, trace-driven simulator using real-world traces, and live-deployments (§ 7). The simulation enables us to understand the system behavior under dynamic conditions, as well as analyze the implications of individual design choices.

6.1 Large Scale Trace Driven Simulations

Datasets: To simulate client activity, we use data from five sources: (i) *GlobalCDN* comprises 8.2M requests sampled from web and video services from 3 GlobalCDN PoPs (two in N. America, one in Europe) for a duration of 6 hours. Each request is a client fetching an object (e.g., web object, video chunk, etc.) and contains user information (IP prefix, ASN, etc.), observed server metrics (goodput, RTTs, loss rates etc.), CDN logs (e.g., user to edge PoP mapping [26, 119]) and performance metrics (time-to-last-byte). (ii) *CAIDA* [22], packet traces from the Equinix data-center in Chicago (in 2016). (iii) *MAWI* [8], packet traces from the WIDE backbone in Japan (in 2017). (iv) *FCC* [41], a U.S. nation-wide home broadband dataset. (v) *Pantheon* [141, 142], a data set of client sessions across different regions.

⁵(e.g., entropy vs Gini impurity for information gain, number of leaf nodes ranging from 50 to 500, ID3, C4.5, and CART for D-Tree)

Generating client sessions: We use our traces to characterize the network conditions of real-world users. CAIDA and MAWI traces are captured at a vantage point between the client and server and we measure the goodput, RTT and loss rate by sequence-matching the data packets with their ACKs⁶. GlobalCDN⁷, FCC and Pantheon datasets include the end-to-end network characteristics between a client and server. We model a client session as a time series of bandwidth (goodput), latency and loss rate measured between a pair of end-points.

Configuration Rewards: To avoid the pitfalls of trace-driven simulations [11], we decouple the modeling of configuration rewards (i.e., PLT calculation) from the process of generating client sessions. Our testbed comprises a cluster of 16 Linux servers (kernel 4.20), divided evenly to act as server (Apache) and clients (Chromium [50]). Our control over the machines and network enable us to set arbitrary server-side configurations (from Table 1) and emulate the bottleneck link to match the measured goodput, latency and loss rates from the datasets (using NetEm, TC [54]), with buffer-size set to Bandwidth Delay Product (BDP). To isolate the impact of network and configuration on PLT, caching (server or browser) is disabled and each server serves a single client (no resource contention). Using this testbed, we exhaustively measure the PLT for all combinations of configurations (Table 1). For each {network condition, configuration} pair, each website is loaded multiple times with the browser⁸. The final results are stored in a large tensor that maps {network condition (goodput, RTT, loss-rate), configuration, website} to PLT – called the *PLT-Tensor* comprising data from the pageloads in the testbed.

Simulator (Virtual Browser): Leveraging the client sessions and the PLT-Tensor, the simulator simulates the client’s browsing behavior and interaction with Configanator as follows (visualized in Appendix F.1): (i) website⁹, user information (e.g., IP) and session characteristics are taken as inputs, (ii) the learning framework determines the appropriate configuration for a connection, and (iii) pageload is simulated by using the PLT-Tensor to determine PLT for the client given the selected configuration. The simulator feeds the PLT back to the learning framework to complete the feedback loop.

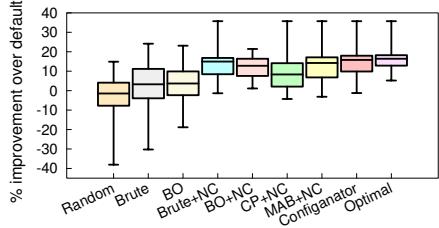
PLT is sensitive to a myriad of features, ranging from dynamic network conditions at different time-of-the-day [67], user devices, to inherent variability. The session time-series captures the network dynamics and the testbed isolates the impact of network conditions on configurations. Table 4 lists the set of knobs we leveraged to test various realistic design choices, e.g., *NCSpread* to test various “K” sizes, *PerfMemory* to test the impact of PLT variability, etc. To account for the other factors like user devices, we conducted a scaled-down

⁶Over a 5-second window (tunable through *ChunkSize* parameter in the simulator), e.g., data ACKed in 5s is used to measure goodput between vantage point/user. Further, we ignore duplicate ACKs while measuring RTTs.

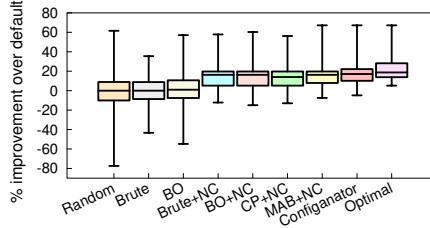
⁷Measurements are on a per-request granularity. For multiple readings within the 5s window, we aggregate and use the median value.

⁸We repeated each measurement 5 times, similar to [135].

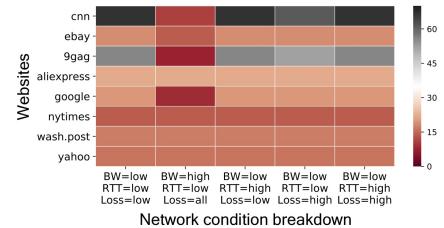
⁹We iteratively load every website from our corpus for a given session.



(a) MAWI traces



(b) GlobalCDN traces



(c) Heatmap of median PLT improvement in different networks (GlobalCDN)

Figure 5: Benefits of Configanator (box whiskers show 5th and 95th percentiles).

experiment on a CDN and tested different configurations for the real-world, diverse user devices (results in § 7.1).

Alternate algorithms: We evaluate against 8 algorithms: **(i, ii) Bruteforce (Brute, Brute+NC):** explores individual configurations, in an online manner, until all are explored and uses the best one (i.e., lowest PLT) for subsequent connections. *Brute* learns at the granularity of individual clients (i.e., unique IP) while *Brute+NC* clusters clients into Network Class (NC), and thus learning is spread across each NC.

(iii, iv, v) Bayesian Optimization (BO, BO+NC, CherryPick+NC): Bayesian Optimization is used to explore the configuration space and the best-explored option is exploited once BO-specific thresholds are met (§ 5). *BO* learns per client and *BO+NC* learns on a user group (NC) granularity. *CherryPick+NC* is similar to *BO+NC* but with hyper-parameters specified in [4].

(vi) Multi-armed Bandit (MAB+NC): uses traditional MAB with a weighted epsilon-greedy agent [133]. Each arm of the bandit is a different configuration, tested on NC granularity.

(vii) Random: Randomly selects a configuration in each trial.

(viii) Optimal: An oracle suggests the optimal parameters for a session by offline brute-force, i.e., PLT is calculated for the entire configuration space for each session offline and the configuration with the lowest PLT is used. This process is repeated for every session and puts an upper bound on improvement.

6.2 Effectiveness of Configanator

Figures 5a, 5b present the improvement in PLTs over default configurations for the different algorithms. The box plots compile data across the website pageloads for the client sessions in the respective trace. Configanator outperforms all alternatives at median and tail, improving p95 PLTs by 67% for GlobalCDN (1500ms), 36% (1100ms) for MAWI, 32% (610ms) for FCC, 48% (640ms) for CAIDA and 57% for Pantheon (850ms). Unlike Default, while Bruteforce and BO apply different configurations to users, they assume that the network remains static and are unable to adapt to fluctuations. Moreover, due to its inability to adjust to fluctuations, BO often explores over 90% of the space without achieving the target EI, behaving similarly to *Brute*. Brutef+NC, BO+NC and CherryPick+NC improve over the prior by amortizing the costs of learning but fail to adjust to non-Gaussian variations.

Although *MAB+NC* is able to handle non-Gaussian noise, it explores/exploits on a per-NC basis and, due to the lack of a cross-NC exploitation arm (Configanator’s DT), *MAB+NC* falls short in its ability to apply patterns learnt across NCs.

As Configanator continuously learns and tests new configurations in an online fashion, a ‘bad’ configuration may be tested during the exploration phase and may lead to performance degradation. This behavior contributes to the worse PLT than Default for the p5 pageload in Figures 5a, 5b. A breakdown of the performance degradation and its causes are presented in Appendix F.8.

Dissecting Performance Improvements: Next, in Figure 5c, we analyze performance breakdown for a subset of websites according to the networking conditions used in prior work [135]). We make two observations: (i) Improvements tend to be higher in low bandwidth, low to high RTT/loss networks (typical for developing regions) with a median value of 14-67% compared to 10-25% for high bandwidth. We postulate that this trend is an outcome of the higher focus on developed region networks (typically high bandwidth, low RTT/loss) for the default configuration selection [33]. We observe a similar trend across our traces: GlobalCDN, MAWI and Pantheon traces (p95 RTT in 100-180ms) tend to show higher improvements than FCC and CAIDA (US-based, ~60ms p95 RTT). (ii) the websites with highest benefits tend to be content-rich, e.g., 9gag.com and cnn.com observe >45% and >60% improvement, respectively, for all low bandwidth networks.

6.3 Benefits of Learning Ensemble

Next, we analyze the convergence for the top-3 algorithms from § 6.2 to focus on the aspects of Configanator that lead to better performance. We further split Configanator into two versions to analyze the benefits of its bandits: “NoGP” lacks GP and guided exploration, while “NoDT” lacks the decision tree.

Figures 6 plots the median distance from optimal across all NC and websites, for the first 500 update iterations. The observations are: (i) As data is gathered, Configanator performs better than others because of its ability to blend the benefits of both GP and DT – essentially efficient exploration and effective exploitation (iterations 3-10). Brutef+NC exhaustively explores the complete space before converging to a choice, while *MAB+NC* exploration lacks the guided nature of acquisition

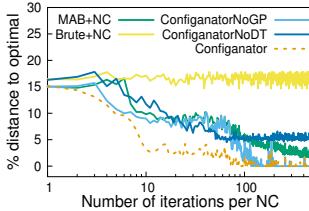


Figure 6: Cold-start convergence to optimal across NCs.

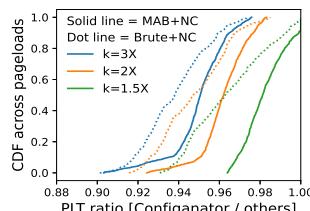


Figure 7: Impact of number of NCs (K) on performance.

function. (ii) Eventually, with sufficient data Configanator-NoGP is able to use the decision tree’s predictive power to achieve near ideal performance (iterations 100+). Although MAB+NC gets within 2-3% of optimal for these iterations, it still needs more iterations to reach the optimal. (iii) While NoGP perform comparably for median at 200+ iterations, performance at the tail is **still** different (Appendix F.4).

6.4 Impact of Network Classes

Impact of Number of NCs: Next, we evaluate the impact of our clustering configuration (i.e., NC_{Spread}) and analyze how the cluster size impacts performance. Intuitively, NC_{Spread} bounds the performance variance within a cluster and has a direct impact on the number of clusters, or ‘ K ’. Given a NC_{Spread} value, the simulator performs a brute-force search to determine the smallest K that yields the threshold. We tested three scenarios with K inflated to $\{1.5, 2, 3\}$ times the baseline value (Figure 5 experiments). The inversion from NC_{Spread} to K and its implications on modeling accuracy are further discussed in Appendix D.

Figure 7 plots the ratio of Configanator and {MAB+NC, Brute+NC} PLT across the pageloads in GlobalCDN trace (<1 when Configanator outperforms). We observe the performance gap between Configanator and others increases with the K size. Although the large K results in a higher number of tighter NCs with lower performance spread within their constituents, it leads to an overall increase in exploration steps for MAB+NC and Brute+NC, as these algorithms explore the individual NCs independently. Further, the individual NC’s best-found configuration is exploited for a narrower set of connections due to a lower number of connections in each cluster as compared to the case when K is small. On the other hand, the DT-arm in Configanator builds on the data collected for *all* NCs (§ 3.1). As soon as Configanator switches to DT-arm fairly early (Appendix F.3), it is able to exploit the best-found configuration for a wider audience, irrespective of the NC boundaries. The higher degree of exploration required by MAB+NC and Brute+NC makes their performance sub-optimal for the NCs with a smaller number of connections. Moreover, this can also lead to performance problems for tail connections, who are often in smaller NC due to their divergent network and device characteristics.

Impact of Size of NC (# of connections): Configanator aggregates network measurement across similar connections

and assumes homogeneity within an NC. Though an NC with a small number of users may lead to a smaller number of connections to learn from, it also favors the system as connections in the respective NC are strictly homogeneous. Next, we explore the impact of this bias on our results. We divide the NCs based on their unique number of IP prefixes and compare the PLTs observed for the individual prefixes with the NC’s global PLT, i.e., median across all the prefixes in the NC. For two of such divisions, Table 3 presents the PLT comparison across the prefixes in NC groups. Compared to the $<=5$ group, i.e., NCs with a small number of distinct prefixes, where performance for most prefixes matches the global one; $>=30$ group shows more varying performance (e.g., lower than global PLT for the p25 prefix). However, we observe that the presence of larger NCs does not drastically impact Configanator as performance for most of the prefixes is still on par with the global one. For the tail prefixes that performs poorly as compared to the global PLT for the $>=30$ group, Configanator overfits the best-found configuration for the NC majority to the tail prefixes, and is observed to still outperform the *Default* (row 4 and 6 in Table 3).

6.5 TCP Connection Reuse (*ConnReuse*)

CDNs typically employ *ConnReuse*, allowing a new request to reuse older TCP connection. The key advantage of this feature is that the new request inherits matured congestion window (*cwnd*) and does not restart the connection from scratch, i.e. ICW. To analyze connection reuse, we analyzed the trace (GlobalCDN) to identify if and when requests reused existing connections and modified our setup to employ the reused connection’s *cwnd* as the ICW for the page load¹⁰.

Figure 8 plots Configanator improvement over *ConnReuse*. We observe that Configanator gains are reduced from 18% over *Default* (Figure 5b) to 14% at the median. The benefits at the tail are still substantial, with 56% p95 improvement. There are several reasons for this behavior: First, connection reuse only impacts the slow-start phase (e.g., ICW) and does not tune the CCA and HTTP, the top two critical knobs (Figure 13). Second, even with reuse, a connection is not always guaranteed to reuse the old *cwnd*, since other TCP settings like *slow_start_after_idle* may reset to default ICW — forcing a reused connection to again go through slow start phase. In fact, the old *cwnd* is reused with a probability of 0.27 in our trace, i.e., only a small subset of requests exploit the benefits of reuse. Third, unlike Configanator’s exploitation of good configurations for similar connections, the scope of *ConnReuse* is limited to a single connection — a new connection from even the same user will go through the default slow start phase. Consequently, while connection reuse outperforms the *Default* by only 4.65% and 19.6% at median and tail respectively, a variant

¹⁰We infer *ConnReuse* if the first *cwnd* for a request is greater than connection’s ICW. Our GlobalCDN trace directly captures these fields for each request. Note that, the reused connection may also inherit the MTU and SRTT values, but we limit our focus to the key component that limit data transfer, i.e., *cwnd*.

NC group	PLT ratio	Prefix distribution				
		p5	p25	p50	p75	95
<=5	Global/Configanator	0.92	1.0	1.0	1.0	1.08
	Default/Configanator	1.05	1.07	1.14	1.27	2.34
>=30	Global/Configanator	0.89	0.96	1.0	1.0	1.06
	Default/Configanator	1.04	1.09	1.17	1.21	2.67

Table 3: Impact of NC size on performance.

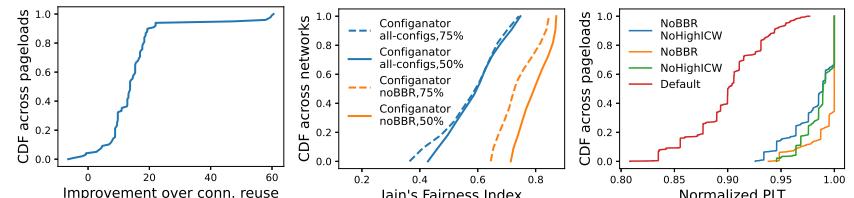


Figure 8: Impact
vs TCP connection reuse.

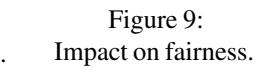


Figure 9:
Impact on fairness.

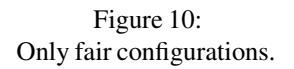


Figure 10:
Only fair configurations.

of Configanator that only tunes *ICW* still performs *ConnReuse* with 8.7% and 23.4% improvements at median and tail.

These results suggest that *ConnReuse* alone is not the silver bullet, also portrayed by other ICW tuning system [42], and Configanator is expected to bring substantial improvements even when traditional optimizations are considered.

6.6 System Benchmarks

Next, we evaluate the latency, CPU and memory overheads. The experiments are performed in a testbed by emulating network conditions from our traces for 10 randomly selected websites. We repeat each test 1000 times.

Latency Overheads: For latency overheads, we focus on the modifications to ALPN to enable HTTP level tuning. We compare Configanator against a version that does not modify ALPN and tunes HTTP level by renegotiates which incurs at least 1-RTT overhead. Figure 11 plots the PLT for the two variants, normalized by *Default* (vanilla Apache). Given that Configanator simply edits the “ALPN next protocol” field in TLS *Server_Hello* without requiring any extra communication, we observe no latency overheads and a similar performance to the *Default*. For *Renegotiation*, we observe a slight PLT inflation ($\sim 3\%$ at the median) which is due to the TLS renegotiation required to switch the HTTP version. We note that this approach still has a minor overhead (4% higher PLT at median) because a page load requires many RTTs and this overhead get amortized.

CPU and Memory Overheads To measure the CPU and memory overheads, we leveraged the Apache Benchmark tool to setup 100, 250 and 500 concurrent connections. We observe slight CPU overhead ($< 5\%$) as compared to *Default*. Although reconfiguring the connections do not require any additional memory, keeping the IP prefixes and their NC/configuration rules in the KV-store contributed to an increased memory usage.

6.7 Fairness Implications

Next, we explore the fairness implications. Within the testbed, we explore the situation where 30 concurrent flows share a representative bottleneck link, i.e., the access links for 3G, 4G, etc (number of flows from [115] Appendix F.10), under shallow buffers ($\{0.5 \text{ and } 1\}$ BDP). We use *Jain’s Fairness Index* [63] to quantify fairness. We split the connections into two groups – one using Configanator and another using the default configuration (e.g., Cubic with 10MSS ICW). We then

vary the percentage of connections in each group.

Quantifying Unfairness: Figure 9(a) present Jain’s index when 75% and 50% of the flows are tuned. We observe that fairness decreases as the percentage of Configanator-tuned flows increases. Unsurprisingly, unfairness arises for two reasons: (1) when a flow is configured to use BBR [24, 57, 111, 129, 137], and (2) when a flow is configured to use high ICW values (even if BBR is not used) [52, 72, 88].

Configanator without unfair configurations: Next, we excluded the unfair configurations from the configuration space and tested 3 scenarios: (i) prevent BBR usage, (ii) prevent high ICW usage, (iii) prevent both. Figure 10 plots the ratio of PLT seen for vanilla Configanator (all configuration) to the variants, for GlobalCDN traces. We observe that NoBBR and NoHighICW perform similar to vanilla system for a significant fraction of the trace (63% and 35% respectively) and within 6% for worst case: this is because BBR is not always the optimal choice and application layer tuning (HTTP version) helps account for the lack of BBR or HighICW.

The results show that Configanator can provide an alternate war-chest to CDNs to improve web performance, even without using the unfair configurations.

6.8 Critical Knobs

We analyze the relative importance of reconfiguring different configuration parameters (Table 1). Our goal is to understand the minimal (or critical) parameters that must be tuned to significantly improve performance. In Figure 13, we plot the performance benefits of using distinct subset of configuration parameters, leveraging the brute-force exploration data from PLT-Tensor. We observe that the top 3 crucial parameters are HTTP version, congestion control algorithm (TCP-CC) and ICW. Moreover, when performing a layer to layer comparison, we observe that the Transport layer parameters combined (*Tran. layer*) have a higher impact on performance than the Application layer knobs combined (*App layer*). To explain this discrepancy, we analyze the different knobs in each layer and we observed that while certain transport knobs, e.g., Auto Corking, have little benefit in the median scenario, they are influential at the tails. Unlike the transport layer, in the Application layer most of the parameters (e.g., HTTP2 settings like header table size etc.) do not show significant benefit in median or tail conditions.

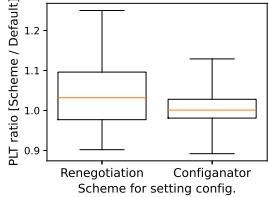


Figure 11:
Latency overheads.

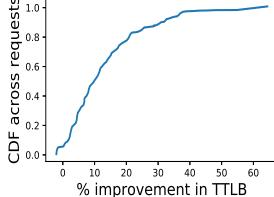


Figure 12: Reconfiguration
benefits for CDN traffic

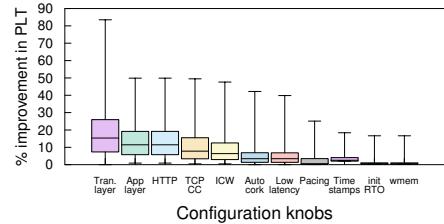


Figure 13: Critical configuration parameters

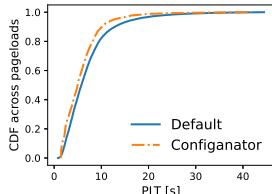


Figure 14: Live deployment PLTs and improvements.

Pageload	PLT diff. (ms)	% imp.
p25	671	13.7
p50	767	14.6
p75	1219	21.5
p95	3797	26.3

7 Live Deployment

In this section, we present the results for dynamically tuning the configurations at scale through a controlled experiment at GlobalCDN and a live prototype deployment on Google Cloud with 3161 end-users.

7.1 Validation at GlobalCDN

Next, we validate our approach when applied to data with more realistic and diverse client settings. We conducted measurements at GlobalCDN to collect data regarding performance of different configurations for the diverse networks and end-user devices. Specifically, we used the *default* configuration for 80% of the connections and explored random configurations for the rest to generate the data needed to emulate the contextual multi-armed bandit based exploration. Due to operational constraints, we analyzed a subset of configuration knobs: different congestion control algorithms and ICW. The experiments were conducted by randomly selecting 1% of the users from 3 of the CDN PoPs, for a duration of 6 hours. Note, these PoPs have the same workloads as the GlobalCDN trace described earlier.

We replayed the captured traces in our simulator, with two key distinctions: (i) the testbed-based PLT-Tensor was replaced by TTLB measurements collected from production users, since these TTLB measurements encompass the performance across real-world users, (ii) This experiment covers diverse user devices in-the-wild. Figure 12 presents the TTLB improvements for Configanator (versus default configuration) with upto 37% improvement at the tail (p95). Although this simulation covers a smaller configuration space, the improvements affirm the efficacy of tuning at scale, working with the diverse set of NC features (user device, network, geo-location, AS).

7.2 Google Cloud Deployment

We deployed Configanator on several Google Cloud servers, each with 8 CPU cores and 32 GB of RAM. We evenly divide

the servers into two groups: one half with the Configanator-enhanced servers, while the other half with traditional Apache server. We cloned a variety of real-world websites from Alexa top-100 and hosted them on servers without sharding. We hosted the Configuration Manager on a dedicated instance.

For clients, we used SpeedChecker [81, 82], a platform for global Internet measurements with vantage points deployed across the globe. We had 3161 clients in total, spread across 4 of the continents. The clients periodically conducted pageloads from both the Configanator and the traditional web servers at the same frequency, resulting in ~150K pageloads in 21 days. Further details about SpeedChecker are provided in Appendix F.1.

Figure 14 plots the raw PLTs observed for the two systems, with the accompanying table summarizing the PLT difference and improvements. Due to the online nature of the exploration and learning, we observe PLT degradation for a small subset of pageloads: 4.3% of the pageloads faced upto -13% degradation. For the rest, Configanator resulted in significant improvements, with upto 3.8s improved PLT at the tail (upto 767ms for the median). Dissecting the improvements across networks and websites, we observe a trend similar to Figure 5c: low bandwidth, high RTT/loss networks and content-rich websites get the most benefits. For the top configurations, we observe no clear winner: top 5 covered 3 CCs (BRR, Cubic and Vegas), both HTTP versions, and ICW ranging from 16 to 40. We observe a stark difference in the ICW values used by clients in developed regions (Europe, N.America), with higher ICW (30-50 MSS), compared to developing regions (16-24 MSS).

Most of the clients (~75%) are from N.America/Europe and the rest are geographically distributed which results in unbalanced Network Classes (NCs), leading to a higher share of traffic for the probes in N.America/Europe. Interestingly, NCs with the most number of pageloads, although showing good improvements (11-13% at median), are not the ones where we observe the highest benefits, owing to their good bandwidth, low RTT connections. We observe that the less-dense NCs still outperform Default (by more than 8% at median), since Configanator’s exploitation arm is able to generalize to a modest extent by using data collected across all NCs.

8 Discussion and Limitations

Security and Equilibrium: Potential implications of self-learning systems include adversarial attacks [123] or oscillations. We are working to formulate the interactions

between different instances of Configanator (i.e., deployments by different CDNs) as a game-theoretic problem to understand our system at equilibrium.

Management Overheads: Dynamically reconfiguring the CDN’s protocol stack complicates performance diagnosis. We plan to investigate methods for reducing this complexity, e.g., minimizing the number of active configuration combinations. Further, different configurations may vary in their resource-consumption at the CDN edge and we plan to investigate the configuration associated resource-overheads in the future.

Data Bias: Configanator’s data-driven workflow can be impacted by the inherent biases of trace-driven systems [11], e.g., choice of configuration can have an impact on the feedback loop’s decision features. We leave a more comprehensive analysis of biasness to future work.

Testbed Limitations: Owing the lack of cellular connections and devices in the testbed, our simulator is unable to emulate different end-user devices and cellular last-miles. Although the dataset from GlobalCDN covers diverse last-mile connections and devices, we plan to explore systematic approaches to incorporate this diversity in the testbed.

Trace Limitations: While several of our traces capture end-to-end behavior (GlobalCDN, Pantheon, FCC), two of our traces do not. Specifically, CAIDA and MAWI traces are from core router and we recreate end-to-end behavior by matching data with ACKs: this recreation can introduce some imprecision into our latency, loss and BW calculations.

NC Size Bias: As demonstrated in the evaluation, connection homogeneity within an NC (due to small NC size) favors Configanator. This bias is prevalent in two of our traces, FCC and Pantheon (comprising synthetically generated flows). However, this does not hold for the realistic traces (e.g., GlobalCDN) which are mainly focused in the evaluation when discussing the size bias, and still shows improvements over the Default.

9 Related Work

Web Performance Many measurement studies [3, 33, 36, 48, 135] have explored the performance of different networking protocol settings and the impact of tuning on web performance. Our system builds on the observations from these studies: namely that different configurations are required for different network conditions and websites. Web improvement by cross-layer tuning was earlier motivated in [91] (Configanator’s workshop paper) and the present paper builds a practical algorithm and system for tuning the configurations. It further evaluates idea in a wide range of realistic scenarios.

Self-Tuning Systems: Self-tuning systems have been explored within the context of transport protocols [31, 64, 77, 98–100, 113, 138], video [2, 68, 85, 124], databases [32, 56, 131], and cloud systems [4, 13, 78, 147]. While our work shares a similar ideology of exploiting heterogeneity, we differ in our methods for learning optimal configuration and in the domain specific solution for implementing

reconfiguration. While [68, 85] employ similar multi-armed bandits, our bandit generalizes across clusters and includes a Gaussian process to speed up learning. Additionally, while some model relatively static or offline workloads [2, 4, 32, 131], Configanator takes an online approach to tackle network and workload dynamics. Unlike [138] which rely on priori assumptions of the network, Configanator builds a performance model-based on live feedback which allows it to adapt to network dynamics. In contrast with [31, 64, 77, 79, 98–100, 113] which focus on tuning specific aspects of stack, Configanator tunes across a broader set of layers and parameters. Similarly, while these techniques use features from only network, Configanator also incorporates application features (e.g., website).

While Configanator focuses on control over server configurations, others [48, 109] require control over both the servers and the network switches to perform appropriate learning and tuning — applicable to data centers. Others [7, 90] move CCA outside data-path, enabling fast development and portability. Such innovative techniques simplify the design of Configanator by externalizing and simplifying tuning.

CrossLayer Optimizations. We differ from existing cross-layer optimizations [3, 9, 15, 25] which introduce APIs to enable the different layers to communicate and react accordingly the network events. Instead, we externalize the optimization logic and present an interface across the different layers to enable an external entity to configure the different layers which requires a learning algorithm agnostic of applications – a key contribution of Configanator.

10 Conclusion

In this paper, we argue that “one-size-fits-all” approach to configuring web server’s network stack results in sub-par performance for end-users, especially those in emerging regions. Due to the ever-expanding nature of Internet, all end-users do not face similar network conditions. This argument stands in stark contrast to the traditional setup of today’s web serving stacks where a single configuration is used for a divergent set of users.

This paper takes the first step towards realizing heterogeneity and fine-grained reconfiguration in a principled and systematic manner: our system, Configanator, introduces a principled framework for learning better configurations, than the default, for a connection by systematically exploring the performance of different configurations across a set of similar connections. We demonstrate the benefits of Configanator using both a live deployment and a large scale simulation.

11 Acknowledgments

We thank the anonymous reviewers, Michael Schapira (our shepherd), Zachary Bischof, Luca Niccolini, Ranjeeth Dasineni, Huapeng Zhou and Ali Razeen for their invaluable feedback on earlier drafts of this paper. We also thank Janusz Jezowicz from SpeedChecker for granting us access to their platform. This work is supported by NSF grants CNS-1819109, CNS-1814285 and Richard B. Salomon Faculty Research Award.

References

- [1] AHMAD, S., HAAMID, A. L., QAZI, Z. A., ZHOU, Z., BENSON, T., AND QAZI, I. A. A view from the other side: Understanding mobile phone characteristics in the developing world. In *Proceedings of the 2016 ACM on Internet Measurement Conference* (2016), ACM, pp. 319–325.
- [2] AKHTAR, Z., NAM, Y. S., GOVINDAN, R., RAO, S., CHEN, J., KATZ-BASSETT, E., RIBEIRO, B., ZHAN, J., AND ZHANG, H. Oboe: auto-tuning video abr algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), ACM, pp. 44–58.
- [3] AL-FARES, M., ELMELEEGY, K., REED, B., AND GASHINSKY, I. Overclocking the yahoo!: Cdn for faster web page loads. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (2011), ACM, pp. 569–584.
- [4] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI* (2017), pp. 469–482.
- [5] ALTHOUSE, J. Tls fingerprinting with ja3 and ja3s. <https://sforce.co/3kUXKv8>.
- [6] ANDERSON, B. Tls fingerprinting in the real world. <https://bit.ly/313Jnoe>.
- [7] ARASHLOO, M. T., GHOBADI, M., REXFORD, J., AND WALKER, D. Hotcocoa: Hardware congestion control abstractions. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), ACM, pp. 108–114.
- [8] ARCHIVE, M. W. G. T. Packet traces from wide backbone 12/1/17 to 12/7/17. <http://mawi.wide.ad.jp/mawi/>.
- [9] BALAKRISHNAN, H., RAHUL, H. S., AND SESHA, S. An integrated congestion management architecture for internet hosts. *ACM SIGCOMM Computer Communication Review* 29, 4 (1999), 175–187.
- [10] BALAKRISHNAN, H., STEMM, M., SESHA, S., AND KATZ, R. H. Analyzing stability in wide-area network performance. *ACM SIGMETRICS Performance Evaluation Review* 25, 1 (1997), 2–12.
- [11] BARTULOVIC, M., JIANG, J., BALAKRISHNAN, S., SEKAR, V., AND SINOPOLI, B. Biases in data-driven networking, and what to do about them. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), ACM, pp. 192–198.
- [12] BELSHE, M., PEON, R., AND THOMSON, M. Hypertext transfer protocol version 2 (http/2). <https://http2.github.io/http2-spec/>.
- [13] BILAL, M., AND CANINI, M. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC ’17, ACM, pp. 189–200.
- [14] BOJAN PAVIC, CHRIS ANSTEY, J. W. Why does speed matter? <https://web.dev/why-speed-matters/>.
- [15] BRIDGES, P. G., WONG, G. T., HILTUNEN, M., SCHLICHTING, R. D., AND BARRICK, M. J. A configurable and extensible transport protocol. *IEEE/ACM Transactions on Networking* 15, 6 (2007), 1254–1265.
- [16] BROCHU, E., CORA, V. M., AND DE FREITAS, N. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599* (2010).
- [17] BROIDO, A., AND CLAFFY, K. Analysis of Route-Views BGP data: policy atoms. In *Network Resource Data Management Workshop* (Santa Barbara, CA, May 2001).
- [18] BROTHERSTON, L. Fingerprints. <https://bit.ly/3BJfFuK>.
- [19] BRUTLAG, J. Speed matters for google web search, 2009.
- [20] BUTKIEWICZ, M., MADHYASTHA, H. V., AND SEKAR, V. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (2011), ACM, pp. 313–328.
- [21] BUTKIEWICZ, M., WANG, D., WU, Z., MADHYASTHA, H. V., AND SEKAR, V. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *NSDI* (2015), vol. 1, pp. 2–3.
- [22] CAIDA. The caida ucsc anonymized internet traces 2016 dataset. <https://bit.ly/311AVG6>.
- [23] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., ET AL. Bbr: congestion-based congestion control. *Communications of the ACM* 60, 2 (2017), 58–66.
- [24] CARDWELL, N., CHENG, Y., YEGANEH, S. H., SWETT, I., VASILIEV, V., JHA, P., SEUNG, Y., MATHIS, M., AND JACOBSON, V. Bbr v2 a model-based congestion control. <https://bit.ly/3x4bQwx>.

- [25] CHEN, A., SRIRAMAN, A., VAIDYA, T., ZHANG, Y., HAEBERLEN, A., LOO, B. T., PHAN, L. T. X., SHERR, M., SHIELDS, C., AND ZHOU, W. Dispersing asymmetric ddos attacks with splitstack. In *HotNets* (2016), pp. 197–203.
- [26] CHEN, F., SITARAMAN, R. K., AND TORRES, M. End-user mapping: Next generation request routing for content delivery. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 167–181.
- [27] CUI, Y., DAI, N., LAI, Z., LI, M., LI, Z., HU, Y., REN, K., AND CHEN, Y. Tailcutter: Wisely cutting tail latency in cloud cdns under cost constraints. *IEEE/ACM Transactions on Networking* 27, 4 (2019), 1612–1628.
- [28] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM* 56 (2013), 74–80.
- [29] DEV@TRAFFICSERVER.APACHE.ORG. Ja3 fingerprint plugin. <https://bit.ly/3iTg70U>.
- [30] DIGITAL, D. Milliseconds make millions: A study on how improvements in mobile site speed positively affect a brand’s bottom line. <https://bit.ly/3rpm8WP>.
- [31] DONG, M., LI, Q., ZARCHY, D., GODFREY, P. B., AND SCHAPIRA, M. Pcc: Re-architecting congestion control for consistent high performance. In *NSDI* (2015), vol. 1, p. 2.
- [32] DUAN, S., THUMMALA, V., AND BABU, S. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.
- [33] DUKKIPATI, N., REFICE, T., CHENG, Y., CHU, J., HERBERT, T., AGARWAL, A., JAIN, A., AND SUTIN, N. An argument for increasing tcp’s initial congestion window. *Computer Communication Review* 40, 3 (2010), 26–33.
- [34] DUMAZET, E. tcp: provide syn headers for passive connections. <https://lwn.net/Articles/645128/>.
- [35] DUNKELS, A., ET AL. The lwip tcp/ip stack. *lwIP—A LightWeight TCP/IP Stack* (2004).
- [36] ERMAN, J., GOPALAKRISHNAN, V., JANA, R., AND RAMAKRISHNAN, K. K. Towards a spdyier mobile web? *IEEE/ACM Transactions on Networking* 23, 6 (2015), 2010–2023.
- [37] FACEBOOK. Aquila (internet deployment by drone). <https://bit.ly/2V92Adk>.
- [38] FACEBOOK. Network connection class. <https://github.com/facebook/network-connection-class>.
- [39] FACEBOOK. Proxygen: Facebook’s c++ http libraries. <https://github.com/facebook/proxygen>.
- [40] FASTLY. Advanced tcp optimizations. <https://bit.ly/3f2SstA>.
- [41] FCC. Measuring fixed broadband report - 2016. <https://bit.ly/2TAxef8>.
- [42] FLORES, M., KHAKPOUR, A. R., AND BEDI, H. Riptide: Jump-starting back-office connections in cloud systems. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)* (2016), IEEE, pp. 78–87.
- [43] FOUNDATION, L. *Open vSwitch*.
- [44] FOUNDATION, O. S. Openssl alpn callback. <https://bit.ly/3rG5rXh>.
- [45] FOWLER, D. Cdn tuning for ott - why doesn’t it already do that? <https://bit.ly/3i4z7Kr>.
- [46] GANJAM, A., SIDDIQUI, F., ZHAN, J., LIU, X., STOICA, I., JIANG, J., SEKAR, V., AND ZHANG, H. C3: Internet-scale control plane for video quality optimization. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)* (2015), pp. 131–144.
- [47] GARDNER, J. R., KUSNER, M. J., XU, Z. E., WEINBERGER, K. Q., AND CUNNINGHAM, J. P. Bayesian optimization with inequality constraints. In *ICML* (2014), pp. 937–945.
- [48] GHOBADI, M., YEGANEH, S. H., AND GANJALI, Y. Rethinking end-to-end congestion control in software-defined networks. In *Proceedings of the 11th ACM Workshop on Hot Topics in networks* (2012), ACM, pp. 61–66.
- [49] GONG, S., NASEER, U., AND BENSON, T. Inspector gadget: A framework for inferring tcp congestion control algorithms and protocol configurations. In *Network Traffic Measurement and Analysis Conference (TMA)* (2020).
- [50] GOOGLE. The chromium projects. <https://www.chromium.org/>.
- [51] GOOGLE. Quic, a multiplexed stream transport over udp. <https://www.chromium.org/quic>.
- [52] GRIECO, L. A., AND MASCOLO, S. Performance evaluation and comparison of westwood+, new reno, and vegas tcp congestion control. *ACM SIGCOMM Computer Communication Review* 34, 2 (2004), 25–38.

- [53] HELT, J., FENG, G., SESHAN, S., AND SEKAR, V. Sandpaper: mitigating performance interference in cdn edge proxies. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing* (2019), pp. 30–46.
- [54] HEMMINGER, S. Netem - network emulator. <https://bit.ly/36ZXT8k>.
- [55] HERBERT, T. tcp: Socket option to set congestion window. <https://bit.ly/3zDnH6r>.
- [56] HERODOTOU, H., LIM, H., LUO, G., BORISOV, N., DONG, L., CETIN, F. B., AND BABU, S. Starfish: A self-tuning system for big data analytics. In *Cidr* (2011), vol. 11, pp. 261–272.
- [57] HOCH, M., BLESS, R., AND ZITTERBART, M. Experimental evaluation of bbr congestion control. In *Network Protocols (ICNP), 2017 IEEE 25th International Conference on* (2017), IEEE, pp. 1–10.
- [58] HOFF, T. Latency is everywhere and it costs you sales - how to crush it. <https://bit.ly/3x5u3Kb>.
- [59] IETF. Rfc 6298. <https://tools.ietf.org/html/rfc6298>.
- [60] IETF. Rfc 7301 transport layer security (tls) application-layer protocol negotiation extension. <https://tools.ietf.org/rfc/rfc7301.txt>.
- [61] INC., C. Malcolm measuring active listeners, connection observers, and legitimate monitors. <https://malcolm.cloudflare.com/>.
- [62] INC., M. Geoip2 city database. <https://www.maxmind.com/en/geoip2-city>.
- [63] JAIN, R., DURRESI, A., AND BABIC, G. Throughput fairness index: An explanation. In *ATM Forum contribution* (1999), vol. 99.
- [64] JAY, N., ROTMAN, N. H., GODFREY, P., SCHAPIRA, M., AND TAMAR, A. Internet congestion control via deep reinforcement learning. *arXiv preprint arXiv:1810.03259* (2018).
- [65] JEONG, E., WOO, S., JAMSHED, M. A., JEONG, H., IHM, S., HAN, D., AND PARK, K. mtcp: a highly scalable user-level tcp stack for multicore systems. In *NSDI* (2014), pp. 489–502.
- [66] JIANG, J., DAS, R., ANANTHANARAYANAN, G., CHOU, P. A., PADMANABHAN, V., SEKAR, V., DOMINIQUE, E., GOLISZEWSKI, M., KUKOLECA, D., VAFIN, R., ET AL. Via: Improving internet telephony call quality using predictive relay selection. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 286–299.
- [67] JIANG, J., SEKAR, V., MILNER, H., SHEPHERD, D., STOICA, I., AND ZHANG, H. Cfca: A practical prediction system for video qoe optimization. In *NSDI* (2016), pp. 137–150.
- [68] JIANG, J., SUN, S., SEKAR, V., AND ZHANG, H. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 393–406.
- [69] JIN, Y., RENGANATHAN, S., ANANTHANARAYANAN, G., JIANG, J., PADMANABHAN, V. N., SCHRODER, M., CALDER, M., AND KRISHNAMURTHY, A. Zooming in on wide-area latencies to a global cloud provider. In *Proceedings of the ACM Special Interest Group on Data Communication* (2019), ACM, pp. 104–116.
- [70] JOHN B. ALTHOUSE, JEFF ATKINSON, J. A. Ja3 - a method for profiling ssl/tls clients. <https://github.com/salesforce/ja3>.
- [71] KAYSER, B. What is the expected distribution of website response times?
- [72] KOZU, T., AKIYAMA, Y., AND YAMAGUCHI, S. Improving rtt fairness on cubic tcp. In *2013 First International Symposium on Computing and Networking* (2013), IEEE, pp. 162–167.
- [73] KRAUSE, A., AND ONG, C. S. Contextual gaussian process bandit optimization. In *Advances in neural information processing systems* (2011), pp. 2447–2455.
- [74] LEE, Y., AND SPRING, N. Identifying and aggregating homogeneous ipv4 /24 blocks with hobbit. In *Proceedings of the 2016 Internet Measurement Conference* (New York, NY, USA, 2016), IMC ’16, ACM, pp. 151–165.
- [75] LETHAM, B., KARRER, B., OTTONI, G., AND BAKSHY, E. Efficient tuning of online systems using Bayesian optimization. <https://bit.ly/3rBMHIm>.
- [76] LI, L., CHU, W., LANGFORD, J., AND SCHAPIRE, R. E. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web* (2010), ACM, pp. 661–670.
- [77] LI, W., ZHOU, F., CHOWDHURY, K. R., AND MELEIS, W. M. Qtcp: Adaptive congestion control with reinforcement learning. *IEEE Transactions on Network Science and Engineering* (2018).
- [78] LI, Z. L., LIANG, M. C.-J., HE, W., ZHU, L., DAI, W., JIANG, J., AND SUN, G. Metis: Robustly tuning tail

- latencies of cloud systems. In *ATC (USENIX Annual Technical Conference)* (July 2018), USENIX.
- [79] LIU, H. H., VISWANATHAN, R., CALDER, M., AKELLA, A., MAHAJAN, R., PADHYE, J., AND ZHANG, M. Efficiently delivering online services over integrated infrastructure. In *NSDI* (2016), vol. 1, p. 1.
- [80] LOON, P. Balloon powered internet. <https://x.company/loon/>.
- [81] LTD., S. Speedchecker. <https://probeapi.speedchecker.com/>.
- [82] LTD., S. Speedchecker - probe api documentation. <https://bit.ly/2TGwdCu>.
- [83] LU, D., QIAO, Y., DINDA, P. A., AND BUSTAMANTE, F. E. Characterizing and predicting tcp throughput on the wide area network. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on* (2005), IEEE, pp. 414–424.
- [84] LU, T., PÁL, D., AND PÁL, M. Contextual multi-armed bandits. In *Proceedings of the Thirteenth international conference on Artificial Intelligence and Statistics* (2010), pp. 485–492.
- [85] MAO, H., NETRAVALI, R., AND ALIZADEH, M. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 197–210.
- [86] MCKAY, M. D., BECKMAN, R. J., AND CONOVER, W. J. Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics* 21, 2 (1979), 239–245.
- [87] MCQUISTIN, S., UPPU, S. P., AND FLORES, M. Tam-ing anycast in the wild internet. In *Proceedings of the Internet Measurement Conference* (2019), pp. 165–178.
- [88] MO, J., LA, R. J., ANANTHARAM, V., AND WAL-RAND, J. Analysis and comparison of tcp reno and vegas. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)* (1999), vol. 3, IEEE, pp. 1556–1563.
- [89] MUKERJEE, M. K., NAYLOR, D., JIANG, J., HAN, D., SESHA, S., AND ZHANG, H. Practical, real-time centralized control for cdn-based live video delivery. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 311–324.
- [90] NARAYAN, A., CANGIALOSI, F., RAGHAVAN, D., GOYAL, P., NARAYANA, S., MITTAL, R., ALIZADEH, M., AND BALAKRISHNAN, H. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), ACM, pp. 30–43.
- [91] NASEER, U., AND BENSON, T. Configtron: Tackling network diversity with heterogeneous configurations. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)* (Santa Clara, CA, July 2017), USENIX Association.
- [92] NASEER, U., AND BENSON, T. InspectorGadget: Inferring network protocol configuration for web services. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)* (July 2018), pp. 1624–1629.
- [93] NASEER, U., BENSON, T. A., AND NETRAVALI, R. Webmedic: Disentangling the memory-functionality tension for the next billion mobile web users. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications* (2021), pp. 71–77.
- [94] NEJATI, J., AND BALASUBRAMANIAN, A. An in-depth study of mobile browser performance. In *Proceedings of the 25th International Conference on World Wide Web* (2016), International World Wide Web Conferences Steering Committee, pp. 1305–1315.
- [95] NETRAVALI, R., GOYAL, A., MICKENS, J., AND BALAKRISHNAN, H. Polaris: Faster page loads using fine-grained dependency tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), USENIX Association.
- [96] NETRAVALI, R., SIVARAMAN, A., DAS, S., GOYAL, A., WINSTEIN, K., MICKENS, J., AND BALAKRISHNAN, H. Mahimahi: Accurate record-and-replay for http. In *USENIX Annual Technical Conference* (2015), pp. 417–429.
- [97] NGINX. Nginx reverse proxy. <https://bit.ly/3yapgbH>.
- [98] NIE, X., ZHAO, Y., CHEN, G., SUI, K., CHEN, Y., PEI, D., ZHANG, M., AND ZHANG, J. Tcp wise: One initial congestion window is not enough. In *Performance Computing and Communications Conference (IPCCC), 2017 IEEE 36th International* (2017), IEEE, pp. 1–8.
- [99] NIE, X., ZHAO, Y., LI, Z., CHEN, G., SUI, K., ZHANG, J., YE, Z., AND PEI, D. Dynamic tcp initial windows and congestion control schemes through reinforcement learning. *IEEE Journal on Selected Areas in Communications* 37, 6 (2019), 1231–1247.

- [100] NIE, X., ZHAO, Y., PEI, D., CHEN, G., SUI, K., AND ZHANG, J. Reducing web latency through dynamically setting tcp initial window with reinforcement learning.
- [101] OF SHEFFIELD, M. L. G. U. Gpyopt. <https://github.com/SheffieldML/GPyOpt>.
- [102] OF SHEFFIELD, M. L. G. U. Gpyopt.core.task.space module. <https://bit.ly/3iVDuHf>.
- [103] OREILLY.COM. Bing and google agree: Slow pages lose users. <https://bit.ly/374YGVI>.
- [104] PELIKAN, M., GOLDBERG, D. E., AND CANTÚ-PAZ, E. Boa: The bayesian optimization algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1* (1999), Morgan Kaufmann Publishers Inc., pp. 525–532.
- [105] PI, Y., JAMIN, S., DANZIG, P., AND SHAHA, J. Appatoms: A high-accuracy data-driven client aggregation for global load balancing. *IEEE/ACM Transactions on Networking* 26, 6 (2018), 2748–2761.
- [106] PICOTCP. picotcp. <http://www.picotcp.com/>.
- [107] QUINLAN, J. R. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.
- [108] RUAMVIBOONSUK, V., NETRAVALI, R., ULUYOL, M., AND MADHYASTHA, H. V. Vroom: Accelerating the mobile web with server-aided dependency resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 390–403.
- [109] RUFFY, F., PRZYSTUPA, M., AND BESCHASTNIKH, I. Iroko: A framework to prototype reinforcement learning for data center traffic control. *arXiv preprint arXiv:1812.09975* (2018).
- [110] RÜTH, J., BORMANN, C., AND HOHLFELD, O. Large-scale scanning of tcp's initial window. In *Proceedings of the 2017 Internet Measurement Conference* (2017), ACM, pp. 304–310.
- [111] RÜTH, J., KUNZE, I., AND HOHLFELD, O. An empirical view on content provider fairness. *arXiv preprint arXiv:1905.07152* (2019).
- [112] RYZHOV, I. O. On the convergence rates of expected improvement methods. *Operations Research* 64, 6 (2016), 1515–1528.
- [113] SCHAPIRA, M., AND WINSTEIN, K. Congestion-control throwdown. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), ACM, pp. 122–128.
- [114] SCHOLZ, D., JAEGER, B., SCHWAIGHOFER, L., RAUMER, D., GEYER, F., AND CARLE, G. Towards a deeper understanding of tcp bbr congestion control. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops* (2018), IEEE, pp. 1–9.
- [115] SCHULMAN, A., LEVIN, D., AND SPRING, N. CRAWDAD dataset umd/sigcomm2008 (v. 2009-03-02). Downloaded from <https://crawdad.org/umd/sigcomm2008/20090302 pcap>, Mar. 2009. traceset: pcap.
- [116] SCIKIT LEARN.ORG. *Decision Trees*.
- [117] SERVER, A. T. Tcpinfo plugin. <https://bit.ly/3x8CyEr>.
- [118] SHAHRIARI, B., SWERSKY, K., WANG, Z., ADAMS, R. P., AND DE FREITAS, N. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE* 104, 1 (2015), 148–175.
- [119] SHUFF, P. Building a billion user load balancer. USENIX Association.
- [120] SINGH, S., MADHYASTHA, H. V., KRISHNAMURTHY, S. V., AND GOVINDAN, R. Flexiweb: Network-aware compaction for accelerating mobile web transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (2015), ACM, pp. 604–616.
- [121] SOFTWARE, V. Varnish http cache. <https://varnish-cache.org/>.
- [122] STEIN, M. Large sample properties of simulations using latin hypercube sampling. *Technometrics* 29, 2 (1987), 143–151.
- [123] SUN, Y., EDMUNDSON, A., VANBEVER, L., LI, O., REXFORD, J., CHIANG, M., AND MITTAL, P. Raptor: Routing attacks on privacy in tor.
- [124] SUN, Y., YIN, X., JIANG, J., SEKAR, V., LIN, F., WANG, N., LIU, T., AND SINOPOLI, B. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 272–285.
- [125] SUNDARESAN, S., FEAMSTER, N., AND TEIXEIRA, R. Home network or access link? locating last-mile downstream throughput bottlenecks. In *International Conference on Passive and Active Network Measurement* (2016), Springer, pp. 111–123.
- [126] TEAM, A. T. R. Bots tampering with tls to avoid detection. <https://bit.ly/3f2cJjb>.

- [127] TLSFINGERPRINT.IO. Tls fingerprint. <https://tlsfingerprint.io/>.
- [128] TRAN-THANH, L., CHAPMAN, A., DE COTE, E. M., ROGERS, A., AND JENNINGS, N. R. Epsilon-first policies for budget-limited multi-armed bandits. In *Twenty-Fourth AAAI Conference on Artificial Intelligence* (2010).
- [129] TURKOVIC, B., KUIPERS, F. A., AND UHLIG, S. Interactions between congestion control algorithms. *network* 3, 17.
- [130] URVOY-KELLER, G. On the stationarity of tcp bulk data transfers. In *International Workshop on Passive and Active Network Measurement* (2005), Springer, pp. 27–40.
- [131] VAN AKEN, D., PAVLO, A., GORDON, G. J., AND ZHANG, B. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 1009–1024.
- [132] VDMS. Our software - cdn. <https://bit.ly/3iOMNbT>.
- [133] VERMOREL, J., AND MOHRI, M. Multi-armed bandit algorithms and empirical evaluation. In *European conference on machine learning* (2005), Springer, pp. 437–448.
- [134] WANG, X. S., BALASUBRAMANIAN, A., KRISHNA-MURTHY, A., AND WETHERALL, D. Demystifying page load performance with wprof. In *NSDI* (2013), pp. 473–485.
- [135] WANG, X. S., BALASUBRAMANIAN, A., KRISHNA-MURTHY, A., AND WETHERALL, D. How speedy is spdy? In *NSDI* (2014), pp. 387–399.
- [136] WANG, X. S., KRISHNAMURTHY, A., AND WETHERALL, D. Speeding up web page loads with shandian. In *NSDI* (2016), pp. 109–122.
- [137] WARE, R., MUKERJEE, M. K., SESHAN, S., AND SHERRY, J. Beyond jain’s fairness index: Setting the bar for the deployment of congestion control algorithms. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks* (2019), pp. 17–24.
- [138] WINSTEIN, K., AND BALAKRISHNAN, H. Tcp ex machina: computer-generated congestion control. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 123–134.
- [139] WOHLFART, F., CHATZIS, N., DABANOGLU, C., CARLE, G., AND WILLINGER, W. Leveraging interconnections for performance: The serving infrastructure of a large cdn. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 206–220.
- [140] WONDRÁ, N. Magic transit: Network functions at cloudflare scale.
- [141] YAN, F. Y., MA, J., HILL, G. D., RAGHAVAN, D., WAHBY, R. S., LEVIS, P., AND WINSTEIN, K. Pantheon datasets. <https://pantheon.stanford.edu/measurements/node/>.
- [142] YAN, F. Y., MA, J., HILL, G. D., RAGHAVAN, D., WAHBY, R. S., LEVIS, P., AND WINSTEIN, K. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association, pp. 731–743.
- [143] YANG, P., SHAO, J., LUO, W., XU, L., DEOGUN, J., AND LU, Y. Tcp congestion avoidance algorithm identification. *IEEE/ACM Transactions on Networking (TON)* 22, 4 (2014), 1311–1324.
- [144] ZEROMQ. Zeromq. <http://zeromq.org/>.
- [145] ZHANG, X., SEN, S., KURNIAWAN, D., GUNAWI, H., AND JIANG, J. E2e: embracing user heterogeneity to improve quality of experience on the web. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 289–302.
- [146] ZHANG, Y., AND DUFFIELD, N. On the constancy of internet path properties. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement* (2001), ACM, pp. 197–211.
- [147] ZHU, Y., LIU, J., GUO, M., BAO, Y., MA, W., LIU, Z., SONG, K., AND YANG, Y. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 338–350.

Knob	Function
TargetAlgo	Sets corresponding tuning algorithm.
TargetNC	Controls the clustering strategy for NCs.
init_samples	Number of samples to initialize an NC.
NCSpread	Controls the performance spread that bounds a NC, and hence the number of cluster (K discussed in § 3.2).
AllowedConfig	Limits the space to disallow certain configurations.
PerfMemory	Length of history for configuration's performance over time.
UpdateLatency	Set latency b/w central <i>Config. Manager</i> and servers.
UpdateFreq	Controls the time after which a model is updated.
ChunkSize	Controls the time window for goodput, RTT and loss-rate measurements from packet traces.

Table 4: Simulator knobs

A Fingerprinting Configurations

Our fingerprinting techniques are inspired from recent works [49, 92, 110, 143]. Our tool inter-operates with TLS and infers configurations in the following ways: (i) HTTP configurations are visible to client during the connection setup and are fingerprinted from the server response, (ii) TCP configurations like RWIN are scraped from the packet headers, (iii) TCP initRTO is measured by emulating a loss during TCP handshake (i.e., by not acknowledging SYN packet back to the server), and measuring the time it takes the server to retransmit the SYN/ACKs, (iv) For TCP ICW, a big enough object URL is scraped from a website, the corresponding object is fetched and the number of packets sent by the server in first RTT is measured. Further, we use MSS=64B to trigger higher number of packets from server. We used AWS in respective regions as the vantage points for fingerprinting the configurations.

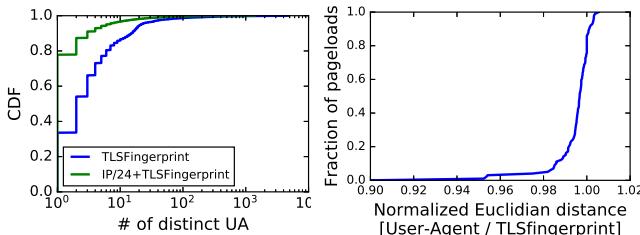


Figure 15:
Relationship between TLS
fingerprint and User-Agent.

Figure 16:
Comparison of device
identifier's impact on NCs.

B TLS Fingerprinting for Device Identification

Recall that instead of the traditional User-Agent string, Configanator uses TLS fingerprinting for device identification as it allows device inference in early stages of the connection (prior to the HTTP version negotiation through ALPN). To evaluate its efficacy, we leverage a dataset from GlobalCDN, comprising 3.6M requests. The dataset consists of server logs and captures User-Agent strings from HTTP GET requests and the TLS fingerprint of the respective connections. The

dataset includes 14.5K unique User-Agent strings and 3.2K unique TLS fingerprints.

Figure 15 plots the number of unique User-Agents (UA) that map to a TLS fingerprint. Ideally, a single UA should map to a fingerprint, thereby accurately identifying the corresponding device. However in practice, we observe that the one-to-one mapping is limited only to 34% of the fingerprints, with the rest mapping to atleast 2 UA. We observe that complementing the TLS fingerprint with the end-user IP-prefix helps in improving the accuracy, with 78% of the IP/24 and TLS fingerprint mapping to a single UA and 96% mapping to at-most 8 unique UA. We observe that for the cases where a single fingerprint maps to multiple UA strings, there are only minor differences, e.g., different browser versions, difference in OS's minor version (Android 6.0 vs 6.1.1).

In Figure 16, we further compare the two device identification techniques for clustering similar connections together. Using a dataset of 89K PLT measurements from GlobalCDN, we run our Network Class clustering using either User-Agent or TLS fingerprint as the basis for device identification. We compute the Euclidean distance of each connection PLT from its cluster's center and the figure plots the ratio of the distance. We observe that the ratio is between 0.98 and 1.00 for the overwhelming majority of the pageloads, indicating that the two technique perform fairly similar. Hence, device identification through TLS fingerprinting provides nearly similar accuracy to the User-Agent strings, with the added benefit that the device is identified prior to negotiating the HTTP version, whereas User-Agent string can only be inferred through HTTP requests headers (received after HTTP version negotiation).

C Passively Recording Network Conditions

Configanator passively collects goodput and packet loss rates for the IP-prefix (/24) and builds a historical archive (§ 3.2). When an IP connects, Configanator uses the handshake RTT and looks-up the goodput and packet loss rates from the recent session for the IP-prefix (/24) to aid in classifying the user into her Network Class. Configanator prototype uses Apache logs to collect information about user IP, the requested content, content size and download time. Additionally, per-connection TCP statistics are captured through Apache TCP Info plugin [117]. Using this information, Configanator calculates bandwidth (goodput) and packet loss rates on a per IP-prefix (/24) basis. Although we use heuristics for stable goodput and loss calculations, e.g., ignoring small objects, the goodput estimate may still under-estimate actual network bottleneck due to TCP mechanics (e.g., slow start phase). Consequently, the use of such measurements in the testbed (§ 6.1) may emulate lower bandwidths and higher loss rates (emulated loss plus induced buffer overflows) than the actual bottleneck links.

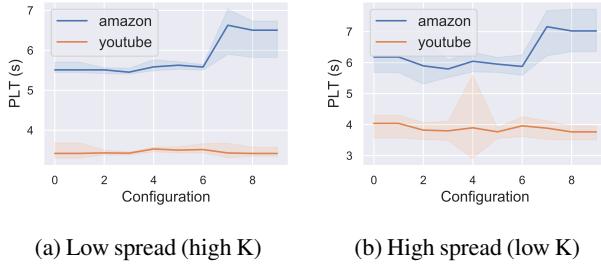


Figure 17: GP config-performance curve.

D Gaussian Process and Network Class Discussion

Bootstrapping GP: The first step of learning is to acquire data to bootstrap the Gaussian process. The bootstrap methodology is crucial for ensuring that the Gaussian-Bandit quickly finds good direction to explore. Recent works [4, 13, 32] have demonstrated the applicability of three distinct bootstrapping approaches: (i) *random*, in which the initial configurations are randomly selected; (ii) *domain-specific*, in which prior domain knowledge, captured through operator interviews or offline simulations, are used to rank configurations to sample; (iii) *Latin Hypercube Sampling* (LHS) which divides the input space into partitions and selects a sample from each partition to spread the samples evenly across space [122]. In this work, we use LHS to bootstrap the learning process. LHS has been found to aid bootstrapping Bayesian optimization by reaching an optimal decision quicker [86]. We observed LHS to speed up exploration in comparison with others by reducing the number of optimization steps by 2-3X, as the bootstrapping samples are spread evenly across space. A perfect rankings of configurations cannot be known prior to actually testing configurations, leading to ranking-based bootstrapping being sub-optimal to LHS.

Individual GP models for each website/Network Class: Bayesian Optimization is traditionally used for mapping configurations to their performance per workload (e.g., cloud configuration to cost [4]). Due to network dynamics and their implications on web performance [67, 135], a separate BO/GP model is required to map configuration performance for each workload (network condition and website), leading to individual exploration for each workload. The lack of cross network/website exploitation (due to separate BO models) makes a solely BO-based technique unfit for Configanator. Intuitively, the system should be able to generalize across networks and can use the already learnt pattern from other networks to a new network, e.g., HTTP/1.1 is optimal at high RTT, high loss for a complex website, no matter the bandwidth [135].

Figure 17 presents the GP model for two websites for the same set of configurations (x-axis) and the same Network Class (NC). In Figure 17a, while GP has estimated the curve for *youtube* with high confidence, *amazon* requires more data samples (wide confidence interval for configuration 7, 8 and

9). The different configuration-performance curves require a separate GP model for each website/NC for correct modeling of a configuration’s performance and effective exploration, as the configuration-performance curve is distinct for every website and Network Class.

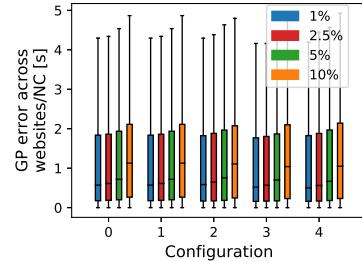


Figure 18: GP modeling error for different NC spread thresholds.

Impact of performance spread within NC: Next, in Figure 18, we leverage the *NCSpread* knob in simulator (Table 4) to test different bounds for Network Classes (NC) clustering. Recall that *NCSpread* controls the “K” for Kmeans clustering by selecting the lowest K that bounds the standard deviation of PLTs for a cluster’s constituents within a specified threshold ({1, 2.5, 5, 10}% in the Figure 18). Determining the right K involves iterating through K values and is a three step process: (i) NC features – network characteristics (bandwidth, latency, loss rate), AS information (ASN, geo-location), and device type – from past connections are clustered using a given K, (ii) For each cluster, the list of PLTs observed for its members connections is generated and is normalized by the median PLT of the list¹¹, (iii) The standard deviation for each list is computed and, based on how far is it from the median and the *NCSpread* limit, the decision to converge on the given K or test a different K is made.

Figure 18 uses the testbed generated data from § 6 and plots the error in GP’s estimate for five randomly selected configurations. The error is calculated as the absolute difference of GP’s PLT estimate for a configuration and the actual PLT, and the boxes plot the error distribution observed for the various clusters (corresponding to the *NCSpread* value) and the websites. Note that, a small error is always expected due to the inherent variability with PLT measurements. The 5% limit *NCSpread* performs fairly close to the lower bounds, while also requiring a lower K: 7% lower K value than the 1% *NCSpread* threshold. This analysis serves as the motivation for using 5% value in the simulator.

Figures 17a and 17b further visualizes the confidence intervals for the GP models for 2 websites. For the high spread case (10% *NCSpread*), connections from slightly different

¹¹As there might be multiple websites, there is one list per website. Further only PLTs for default configuration are used.

networks are mapped to the name GP, resulting in wider confidence intervals, and leads to inefficiency with acquisition function’s next configuration suggestion.

E Deployment Considerations

Data-driven systems [2, 46, 66–68] traditionally use a split-plane architecture where a modeling layer (responsible for ingesting huge amounts of data and updating models) runs at a slower granularity than the decision layer (responsible for applying modeled decisions for users at real-time). Configanator’s architecture uses a split-plane model which leverages the different computational requirements of Configanator’s workflow: As demonstrated in Figure 4, in the slow path, the Configuration Manager collects telemetry from the web servers, uses this telemetry to update the learning model, and installs the configuration rules created by the model into the web servers. In the foreground, each web server uses the pre-installed rules to apply configuration to each connection and periodically collects telemetry from each connection.

The *first* phase, the background process, is time-consuming because of the process of updating the learning algorithms and Network Classes. The *second* phase, a fast, real-time process that applies the configuration rules to each inbound *user connection*, is run at the edge on each web server and provides low-latency, dynamic tuning. We note that although this decoupling results in the fast-path using stale information, we observe that this stale information still provides near-optimal performance [46].

F Supplementary Evaluation Material

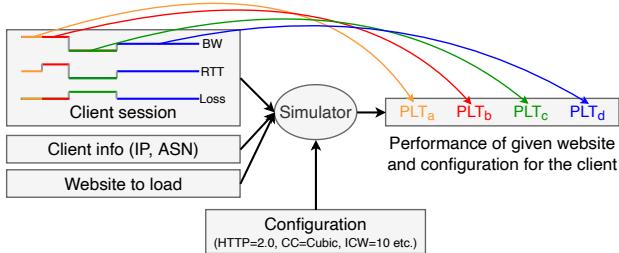


Figure 19: Simulating pageload for a client

F.1 Evaluation Setup

Simulation workflow: Figure 19 presents the workflow for simulating pageload performance. The client sessions are extracted from the real-world datasets and are modeled as time-series. Since we use 5s for measuring the network characteristics from the trace (a tunable knob as discussed in § 6.1), each linear state for BW, RTT, Loss in Figure 19 is at

least 5s long. We extract an IP distribution from the trace to model the temporal aspects of client’s connections (time at which a client connection (or IP) is seen in trace), i.e., the user sessions are fed to the simulator in the order they are observed in the real-world trace.

The simulator takes the goodput, RTT, loss rates at a certain time from the session time-series, client info (IP, ASN) and the target website to load as input. Using these features, it consults the configuration to test from the learning framework. Once the target configuration is known, it leverages the PLT-Tensor to map the network characteristics {goodput, RTT, loss-rate}, website and configuration to the eventual PLT. Note that, we assume that the network characteristics stay stable throughout the lifetime on a single pageload, supported by recent studies that TCP connection is piece-wise stationary and each segment stays stable in the order of tens of seconds to minutes [10].

Table 4 further summarizes a number of simulator knobs that allow us to emulate and test a variety of scenarios.

Dataset description and breakdown: While the Global-CDN, MAWI and CAIDA datasets are adequately described in § 6.1, here we provide details for the other two datasets.

The Pantheon dataset [141, 142] comprises of synthetically generated TCP flows across the different parts of the world. We collected three month’s worth of data (May to July 2018) from Pantheon’s website [141]. For the generated flows, the dataset logs the flow IDs, packet ingress/egress timestamps, packet sizes and one-way delay. Using these fields, we calculate the goodput, RTT and loss rates between each pair of end-point and, similar to the case for GlobalCDN, MAWI and CAIDA datasets, generate the time-series for the network characteristics. These end-points (vantage points) range from AWS deployments to university networks and cover multiple last-mile connection types. The FCC dataset is collected by the Measuring Broadband America program [41] and consists of a nation-wide study of end-user’s broadband performance and an accompanying dataset. This dataset provides coarse granularity measurements in form of bandwidth, latency and loss rates distributions measured for real-world users. We use these distributions to generate synthetic traces, similar to [2].

The breakdown of $\sim 21.4M$ sessions is as follows: 8.2M from GlobalCDN, 2.7M from MAWI, 8.1M from CAIDA, 1.6M from FCC, 800K from Pantheon. The cross-regional nature of our datasets provide coverage over a wide range of representative network conditions, e.g., while FCC and CAIDA cover connections in U.S., MAWI dataset is from East Asia. Further, GlobalCDN and Pantheon [142] are even more diverse with connections from countries across the globe.

SpeedChecker and vantage points: SpeedChecker [81] is a platform for global Internet measurements, with vantage points deployed in over 170 countries and thousands of ISPs. SpeedChecker provides an API to conduct automated measurements ranging from ping, DNS, web pageloads to video tests. We leveraged vantage points (windows machines) on this platform for conducting the pageloads. The API call

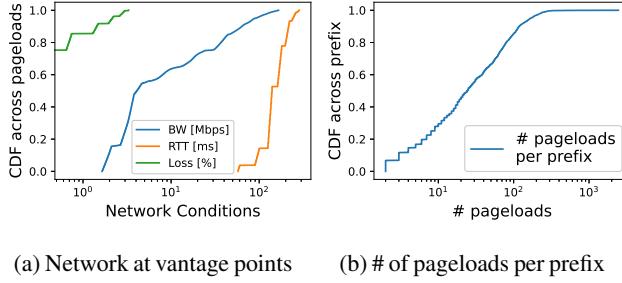


Figure 20: Live-deployment vantage points

requires *CountryCode* and *Destinations* (a list of URLs to load). Vantage points (probes) from the specified country are selected internally by their platform and pageloads are conducted (upto 100 pageload every hour, per city in the country). Figures 20 presents various distributions about our vantage points. 20a compiles the distribution of network conditions observed for each pageload. The vantage points vary across the three dimensions and have mostly RTTs greater than 100ms. 20b presents the number of pageloads per prefix. We observe a heavy tail distribution, where certain vantage points conducted more pageloads than others, e.g., Europe, N.America had 4X more pageloads than Asia and Africa due to the higher number of the SpeedChecker clients in the developed regions. Africa had the smallest number of vantage points among all continents and the hourly limits were frequently reached, resulting in a lower number of total pageloads. Note that, diverse network conditions were still observed for the vantage points (Figure 20a) in spite of this skew in vantage point location. We further observe that 90% of the vantage points have unique IP-prefix (/24), showing that they are distributed and are not placed in a single facility, in the same subnet.

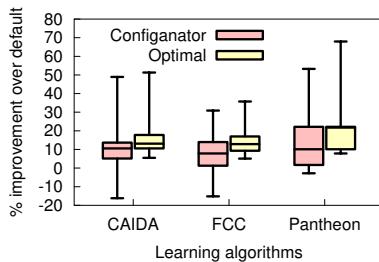


Figure 21: Configanator performance for CAIDA, FCC and Pantheon traces

F.2 Configanator Performance for CAIDA, FCC and Pantheon Traces

Figure 21 presents the distributions for Configanator's PLT improvement for the CAIDA, FCC and Pantheon traces. These figures complement the results in § 6.2 where we could not add

the results for all the traces due to space limitations. CAIDA and FCC traces are collected from U.S.A and mostly cover high bandwidth, low RTT/loss connections, e.g., p95 RTT is 60ms. Following the trend observed in Figure 5c, we observe their PLT improvements over default to be lower as compared to other traces. Especially FCC dataset covers broadband connections and we observe the lowest p95 PLT improvement for FCC among all the datasets. Nevertheless, the improvements are still substantial with 610-640ms decrease in p95 PLT. On the other hand, Pantheon traces cover wider range of networks, often across continents, and result in upto 850ms improvement at tail.

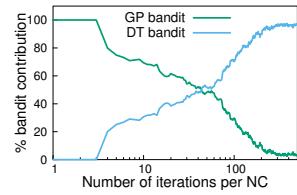


Figure 22:
Bandits contribution

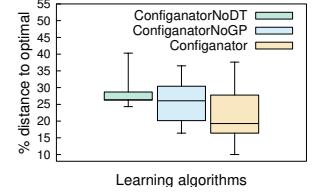


Figure 23: Tail performance

F.3 Bandit Contribution

Figure 22 uses the same convergence analysis as Figure 6 and plots the percentage of connections that uses a certain bandit. Initially GP bandit is largely used for a guided exploration. However, as more data is collected, DT bandit starts to overshadow the GP bandit, highlighting that a per-NC guided exploration is over-shadowed by cross-NC exploitation, when large data is available.

F.4 Bandit Performance at Tail

Figure 23 focuses on tail by dividing the entire trace into one minute segments and plotting the distance to optimal for the worst-case tail of each minute. Configanator's use of bandits enables it to perform better than individual bandits, being closer to optimal by more than 7%.

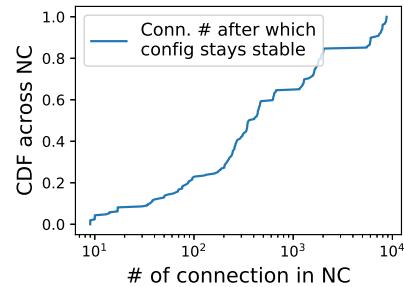


Figure 24: Time of last config. change in NC

F.5 Configuration Stability

Figure 24 plots the number of connections across NCs after which the DT-bandit’s decision stays stable, i.e., configuration decision for the NC does not change. While for the median NC, the configuration choice becomes stable at ~ 400 th connection; we observe that it can take as much as 10K connections to reach the final configuration for some NCs. We observe the DT-bandit to stuck on a near-optimal configuration for these NCs. Down the line, the epsilon-bandit, randomly exploring, finds the optimal configuration and updates the NC. We note that Configanator switches to DT-bandit in the first 10-15 iterations for these NC, highlighting that the GP model’s EI threshold was reached very early, and the initial exploration through GP was not very beneficial in uncovering the optimal configuration.

F.6 Design Choices for Network Classes

We use *GlobalCDN* dataset to evaluate design choices for classifying similar users together. We compare Configanator’s clustering with: (i) *IP-Prefix* clusters /24 users together, (ii) *Hobbit* [74] improves /24 groups by merging dis-contiguous /24s based on co-location in Internet topology and homogeneous performance, (iii) *Latency Driven* inspired from AP-Atoms [105] where users with similar latency are grouped together, (iv) since CDNs group users based on their performance similarity [26, 119], *CDN Aggr.* use the natural CDN grouping and assigns all users mapped to a PoP to the same NC. We extract these mappings from the GlobalCDN dataset and, as these mappings can vary over time, build a time-series of user to CDN PoP mapping.

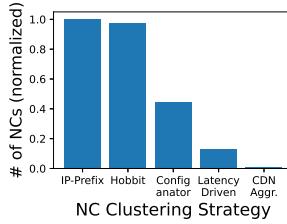


Figure 25: Impact of clustering on # of NC

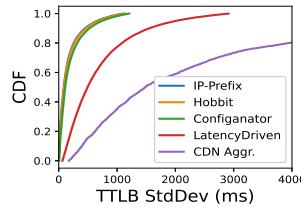
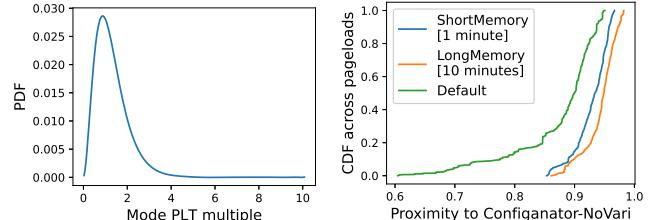


Figure 26: Spread of TTLBs within NC

Figure 25 and 26 plots the number of NCs and the spread of TTLBs within an NC for the different strategies. Ideally, Configanator favors small number of NCs and aims for small to negligible variations within NC performance metric, as the goal is to cluster similarly performing users together (§ 3.2). We observe *Hobbit* subnets /24 groups to have a poor coverage over the trace (*Hobbit* only covers 12% of prefixes in Global-CDN dataset), with non-*Hobbit* /24s being treated as individual groups, leading to similar results as *IP-Prefix* (figure 25). Although NCs built by *Hobbit* and *IP-Prefix* have lowest performance divergence, (low std. dev. in figure 26); Configanator NCs are almost similarly compact, while using less than half



(a) Observed PLT variations.

(b) Impact of PLT variability.

Figure 27: PLT variability

number of NCs. Although *Latency Driven* uses least number of NCs, the lack of device, bandwidth, loss etc. information leads to diverse users being grouped together (high TTLB std. dev.). Similarly, since CDNs maps user based on latency to their closest PoPs, network and device heterogeneity still exist (e.g., the closest PoP to a user can be 10ms-600ms [26]), leading to highest performance variation within an NC for *CDN Aggr.*

We further modified the simulator to explore Configanator performance when different NC techniques are used. We observe that Configanator out-performs the rest for the majority of the pageloads. The prefix and CDN based approaches either do not account for network dynamics or overfit to specific regions respectively. Latency driven performs slightly better but ignoring the important metrics, like packet loss and bandwidth, degrades its effectiveness.

F.7 PLT Variability

PLT measurements are inherently noisy [96] and the variability in PLT can disrupt the learning algorithm’s model, e.g., GP is sensitive to noise [78]. Using data from a web performance observability company (NewRelic [71]), we modeled PLT variability distribution and used it to introduce variability in testbed-generated PLT-Tensor. Figure 27a plots a PDF of the variations with x-axis as the mode PLT multiple (x-axis is PLT normalized by mode PLT). We fit an Erlang curve to the observed PDF, owing to its right-skewed, long-tail nature. Using PLT from PLT-Tensor as the mode PLT (since mode PLT is the most stable PLT measurement), the PDF is used to calculate the noisy PLT observed by a real-world user.

Figure 27b plots the extent to which Configanator decisions (in face of PLT noise) are optimal, compared to the case when there is no noise (Configanator-NoVari). A proximity score of 1 indicates that Configanator decisions stay exactly the same for both (noise, no-noise) cases. Leveraging the *PerfMemory* knob, we test 2 scenarios with different length of historical memory. Configanator uses this historical memory to amortize the impact of any sudden change in performance metrics. We observe Configanator’s decisions to slightly deteriorate in face of noise. However the extent is mild at worst — with the system still assigning the optimal decisions more than 95% at median.

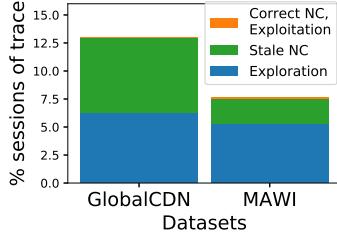


Figure 28: % sessions with PLT degradation.

F.8 Dissecting PLT Degradation

As shown in Figure 5, all algorithms result in some PLT degradation. Figure 28 plots the percentage of sessions that faced PLT degradation and further divide them into the root-causes. Our observations are as follows: (i) During exploration, multiple configurations are tested and may result in degradation. Around 5-6.2% of the sessions in two of the datasets are such exploration steps. (ii) As network conditions change over time, Configanator’s estimate of historical network characteristics for an IP-prefix may diverge from the actual network. The stale information is used for classifying the connection into an NC and predicting the optimal configuration. Due to the global nature of GlobalCDN dataset, we observe a higher network churn, with 6.6% of total sessions resulting in PLT degradation due to stale NC. Only a small proportion of sessions in MAWI dataset ($\sim 0.1\%$) resulted in PLT degradation with correct NC view, indicating that the exploitation arm momentary got stuck at a sub-optimal configuration.

F.9 CM Design Choices

Configuration Manager Design: CM can run locally in a PoP or centrally within a data-center [46], trading-off between data-size to learn and the speed to react to changes. We evaluate both scenarios in our simulator: In the local design, there’s a separate CM for each trace, while for the global case there’s a single CM for all traces. To simulate each scenario, we vary the latencies between CM and the web servers. We observe that while the global CM is able to make slightly better predictions at the tail (2% better than local), the difference at median is

¹² It takes ~ 2 minutes to update the models for 10K sessions.

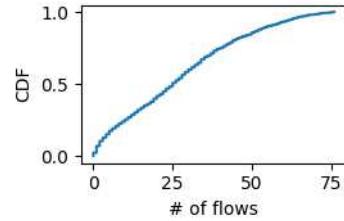


Figure 29: Number of flows through access link

negligible. Despite the larger data set, global CM is not significantly better due to distinctly diverse network conditions across regions (only 17% NCs are common in U.S and Japan traces).

Frequency of model updates: Next, we analyze the impact of updating our performance model less frequently: we explore a range of values from every 2 minutes ¹² up to every day. We observed performance to stay relatively stable at the median, whereas hourly or lower update intervals result in $\sim 8\%$ better improvement at tail, than a per-day granularity.

F.10 Flows Through Access Link

We use packet trace from [115] to measure the typical number of TCP flows through an access link. Figure 29 presents the number of TCP flows with at least 10Kb data transferred, in a 60s time interval. On the median 60s time interval, we observe around 25-30 flows competing through the access link.

F.11 Additional Micro-benchmarks

In addition to the system benchmarks in § F.9, we also evaluated two alternate design choices: VMs and LD_Preload. For VMs, we used one VM for each configuration and used Open vSwitch (OVS) [43] for routing flows to the appropriately configured VM. We explored the use of *LD_Preload* to intercept system call and tuned socket using *setsockopt()*. In comparing both choices with Configanator, we observed that the VM-based approach introduced a 20% increase in latency where as the *LD_Preload* introduced a much smaller latency of 2.2%. We also observed overheads for CPU and Memory utilization: the VM-based approach introduced 30% (memory taken by the guest OS) while *LD_Preload* introduced a 5% increase.

C2DN: How to Harness Erasure Codes at the Edge for Efficient Content Delivery

Juncheng Yang¹, Anirudh Sabnis², Daniel S. Berger³, K. V. Rashmi¹, Ramesh K. Sitaraman^{2,4}

¹Carnegie Mellon University

²University of Massachusetts, Amherst

³Microsoft Research and University of Washington

⁴Akamai Technologies

Abstract

Content Delivery Networks (CDNs) deliver much of the world’s web and video content to users from thousands of clusters deployed at the “edges” of the Internet. Maintaining consistent performance in this large distributed system is challenging. Through analysis of month-long logs from over 2000 clusters of a large CDN, we study the patterns of server unavailability. For a CDN with no redundancy, each server unavailability causes a sudden loss in performance as the objects previously cached on that server are not accessible, which leads to a miss ratio spike. The state-of-the-art mitigation technique used by large CDNs is to replicate objects across multiple servers within a cluster. We find that although replication reduces miss ratio spikes, spikes remain a performance challenge. We present C2DN, the first CDN design that achieves a lower miss ratio, higher availability, higher resource efficiency, and close-to-perfect write load balancing. The core of our design is to introduce erasure coding into the CDN architecture and use the parity chunks to re-balance the write load across servers. We implement C2DN on top of open-source production software and demonstrate that compared to replication-based CDNs, C2DN obtains 11% lower byte miss ratio, eliminates unavailability-induced miss ratio spikes, and reduces write load imbalance by 99%.

1 Introduction

Content Delivery Networks (CDNs) [20] carry more than 70% of Internet traffic and continue to grow [19]. Large CDNs achieve this by operating thousands of clusters deployed worldwide so that users can download content with low network latency. When a user requests an object, the CDN routes the request to a server proximal to the user [15]. If the server contains the requested object in its cache, the user experiences a fast response (*cache hit*). If no server within the cluster has the object in cache (*cache miss*), the object is fetched from a remote cluster which could be another CDN cluster or the origin (i.e., the content provider).

Detrimental effects of cache misses. Cache misses have three detrimental effects. First, they degrade performance

by increasing the *content download times* experienced by the user, as each object incurring a cache miss would have to be downloaded over the WAN from a remote server. Second, cache miss can result in additional traffic between the CDN cluster and the origin, which is a significant bandwidth cost for CDN operators. Third, if more cache misses are served from the origin, content providers need to provision more servers with higher network bandwidth. Consequently, a CDN’s goal is to minimize the miss ratio and maintain a low miss ratio over time for all content providers.

Why tail performance matters. The design goal of a CDN is to consistently improve download performance for *all* objects on a content provider’s site, in *every* time window, and for *each* client location. The performance improvement is viewed as a “speedup” that the CDN provides over the content provider’s origin, i.e., it can be quantified as the ratio of the time to download an object directly from origin (without the CDN) to the time to download the same object from the CDN. A CDN’s goal is to provide a significant average speedup in every time window (say, 5-minute window) and at each client location. A spike in the miss ratio in a single cluster could violate these performance goals, *even if that spike is short-lived and impacts only a subset of the objects*. That is because the CDN likely offers no speedup over origin for any client download that is a cache miss, and indeed a short-lived spike in miss ratio could drastically decrease the average speedup provided by the CDN during a 5-minute period.

The challenge of frequent server unavailabilities. Due to stringent performance goals, servers are continuously monitored by the cluster’s load balancer. A server is declared to be “unavailable” and (temporarily) taken out of service if it is deemed incapable of serving content to users within specified performance bounds. By analyzing a month long logs from the load balancers in over 2000 clusters of a large CDN, we find that server unavailability is very common in CDN edge clusters. When a server is unavailable, the objects stored in its cache are not available to serve user requests. Unless the requested objects can be retrieved from other servers within the

cluster, these objects need to be fetched from a remote server, resulting in a spike in the miss ratio, potentially causing a violation of performance guarantees.

Limitations of the state-of-the-art approaches. To tolerate server unavailabilities, the state-of-the-art approach adopted by large CDNs is to replicate objects across two servers within a cluster. We found that this approach has three significant limitations. First, we find that object replication does not eliminate the miss ratio spike following a server unavailability event. The reason is that the replica of the object (within the cluster) may no longer be present due to eviction from its cache. Second, replicating objects is space-inefficient as the CDN effectively has to provision twice the cache capacity, which is challenging due to the accelerating growth in CDN traffic. Third, we observe a significant *write* imbalance between servers due to DNS based load balancing. This imbalance increases SSD read latency and reduces SSD lifetime [22, 61].

Bringing together efficiency and high availability. In this paper, we present C2DN¹, a CDN design that achieves both high availability and high resource efficiency. To achieve high resource efficiency, we apply erasure coding to large cached objects. This requires overcoming multiple CDN-specific challenges such as eviction of object chunks due to write rate imbalances. In fact, we show that a naive application of erasure coding fails to achieve the goal. The core of our design is a new technique that enables CDNs to balance eviction rates and write loads across servers in each cluster. We exploit the fact that erasure coding enables more flexibility in assigning chunks to multiple servers. Our key insight here is that the chunk assignment can be reduced to a known mathematical optimization problem, called Max Flow Problem.

The core contributions of C2DN are a novel chunk placement scheme for consistent-hashing-based load balancing in CDN clusters and a low-overhead implementation of erasure coding for CDNs that can serve the different traffic requirements of production systems. Specifically, by solving an instance of the Max Flow problem, we assign objects with *near-optimal balance* in eviction and write rates for CDN servers and their SSDs. As a consequence, C2DN can reduce storage overheads and bandwidth costs. Finally, equal write rates across servers essentially function as a cluster-wide distributed wear-leveling for the servers’ SSDs, significantly extending lifetimes.

Our contributions. We make the following contributions.

1. We show that server unavailability is common in CDN clusters by analyzing a month-long trace from over 2000 load balancers of a large CDN. We show that the state-of-the-art approach of replicating objects within a cluster does not eliminate miss ratio spikes after a server unavailability events.
2. We design C2DN with a hybrid redundancy scheme us-

ing replication and erasure coding, along with a novel approach for parity placement. C2DN reduces the storage overhead of providing fault tolerance, and hence lowers the miss ratio. Moreover, by leveraging the parity assignment, C2DN balances the write loads and eviction rates across cache servers.

3. We implement C2DN on top of the Apache Traffic Server (ATS) [7] and evaluate it using production traces. We show that C2DN provides 11% miss ratio reduction compared to the state-of-the-art, and C2DN eliminates the miss ratio spikes caused by server unavailabilities. Further, C2DN decreases write load imbalance between servers by 99%.

2 Background

We describe CDN architecture, performance, and cost factors.

CDN Architecture. A CDN is a large distributed system with hundreds of thousands of servers deployed around the world [20, 50]. The servers are grouped into *clusters*, where each cluster is deployed within a data center on the edge of the Internet. The CDN cluster caches content and serves it on behalf of *content providers*, such as e-commerce sites, entertainment portals, social networks, news sites, media providers, etc. By caching content in server clusters proximal to the end users, a CDN improves performance by providing faster download times for clients. Unlike storage systems, CDN servers do not store the original content copies. When the requested content is not available in the cluster (cache miss), the content is retrieved from other CDN cluster or the origin servers operated by the content provider.

Bucket-based request routing. When a user requests an object, such as a web page or video, the *global load balancer* of the CDN routes the request to a cluster that is proximal to the user [15]. Next, the *local load balancer* within the cluster routes the request to one or more servers within the chosen cluster that can serve the requested object. As an example, in Akamai’s CDN, these routing steps are performed as DNS lookups. A content provider CNAMEs its domain name (e.g., for all of its media objects) to a sub-domain whose authoritative DNS server is the CDN’s global load balancer. At the global load balancer, this sub-domain is CNAME’d to a cluster-local load balancer that assigns the sub-domain to a cluster server using consistent hashing [43].

CDN request routing stands in contrast to sharding in key-value caches, such as Memcached and Redis, where consistent hashing is often applied at a per-object granularity [49, 81–83]. In CDNs, *load balancing decisions are taken on the granularity of groups of objects called buckets*. Each bucket, in a DNS-based load balancer, correspond to a domain name that is resolved to obtain one or more server IPs that host objects in that bucket. This resolution is computed using consistent hashing. Since the number of buckets is limited in the range of 100s, the computation is performed and cached when a cluster server becomes available or unavailable.

¹C2DN stands for Coded Content Delivery Network.

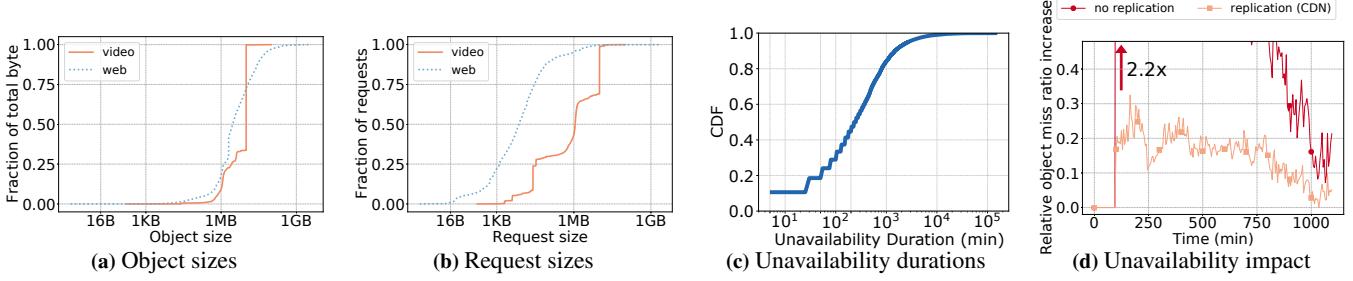


Figure 1: a) Size distributions show that large objects contribute to most of the unique bytes and b) most requests are for small objects. c) Server unavailabilities are mostly transient. d) Object miss ratios spike after server unavailability both with and without the state-of-the-art replication.

CDN Performance Requirements. A CDN aims to serve content faster than a customer’s origin by a specified speedup factor. This factor is commonly part of a service level agreement (SLA) between the CDN and the content provider. The SLA is monitored by recording download times from a globally distributed set of locations for the same content using both CDN and origin servers. Hence, the goal is to ensure good “tail” performance in *every* time interval for *every* content provider from *every* cluster.

Operating Costs of a CDN. CDNs seek to minimize the operating cost, which consists of the following main categories. (i) *Bandwidth*: A major component of the operating cost of a CDN is bandwidth, accounting for roughly 25% of operating costs. The bandwidth cost can be further broken down, the bandwidth cost caused by cache miss traffic called *midgress* [69] that accounts for roughly 20%, while the rest is the cost of *egress* i.e., the traffic from the CDN servers to clients. CDNs have a great cost incentive to reduce the byte miss ratio and the midgress traffic since a CDN gets paid by content providers for the traffic to end users. The midgress traffic between CDN clusters and the origin is purely a cost overhead for the CDN. Even modest reductions in midgress translate into large cost savings since the bandwidth costs tens of millions of dollars per year for a large CDN [69].

(ii) SSD wearout: A second major cost component is server depreciation which accounts for about 25% of the operating cost of a large CDN. Hardware replacements are particularly expensive for small edge clusters due to the large geographic footprints of CDNs. SSDs are a key component due to the high IOPS requirements of CDN caching. Unfortunately, using SSDs in caching applications is challenging due to their limited write endurance [9, 22, 42, 67, 70]. With deployments of TLC and QLC SSDs, reducing SSD write rates has become even more critical. Besides reducing the average write rate within a cluster, CDNs also seek to reduce the variance of write rates of different servers and their SSDs. Large variance leads to some SSDs not achieving their intended lifetime (e.g., 3 years) as well as high tail latency (see §3.4). Consequently, CDNs seek to reduce the peak write rate, ideally balancing write rates across all SSDs in a cluster.

Per server load (TB)	Max	Min	Mean	Max/min
Weekly read	225.2	167.9	191.2	1.3
Weekly write	16.54	6.69	12.57	2.5

Table 1: Read and write load for a 10-server production cluster.

3 Production CDN Trace Analysis

This section motivates the design of C2DN by analyzing three sets of traces from production Akamai clusters.

We collected request traces from two typical Akamai 10-server-clusters (cluster cache size 40 TB), one mainly serving web traffic and the other mainly serving video traffic. These traces comprise anonymized loglines for every request from every server over a period of 7 and 18 days, respectively. The **web trace** totals 6 billion requests (1.7 PB) for 273 million unique objects (79.8 TB). The **video trace** totals 600 million requests (2.1 PB) for 130 million unique objects (224 TB).

Additionally, we collected availability traces from 2190 Akamai clusters over 31 days. The trace consists of snapshots taken every 5 minutes from each cluster’s local load balancer. Each snapshot contains the number of available servers as determined by the load balancer. The smallest cluster has two servers, the largest cluster has over 500 servers, and the median cluster size is 17 servers. We observe that cluster size has a wide range, and around 40% of clusters have fewer than or equal to 10 servers. We plot the distribution of cluster size in Fig. 10 in Appendix 10.1.

3.1 Diversity in workloads and object sizes

CDNs mix different types of traffic in clusters in order to fully use their resources. For example, different “classes” of traffic with small and large object sizes, such as web assets and video-on-demand, are mixed to balance the utilization of the cluster’s CPUs as well as network and disk bandwidth [68]. Consequently, object sizes vary widely [10]. Figures 1a and 1b show the size distribution for our production traces, weighted by unique objects and by request count, respectively. As expected, object sizes vary from a few bytes to a few GBs. Fig 1a shows that *the majority of traffic and cache space is used by large objects*. Furthermore, objects smaller than 1 MB make

up less than 15% and 12% of the total working set in web and video, respectively. Fig 1b shows that *the majority of the requests are for small objects* with 95% of requests in web-dominant workload smaller than 1 MB, and 50% of requests in video-dominant workload smaller than 1 MB.

3.2 Unavailability is common and transient

Unavailability is common. Across all clusters, server unavailabilities occur in 45.2% of the 5-minute snapshots. For clusters with only ten servers (same size as the cluster we collect request traces from), we observe that 30.5% of 5-min time snapshots show server unavailability. Moreover, we observe that unavailability affects only a small number of servers at any given time: 85% of unavailabilities affect less than 10% of servers in large clusters, and 84% of unavailabilities affect no more than a single server in a ten-server cluster.

These unavailability rates can appear high compared to published failure rates in large data centers [25, 46, 54, 56, 59] and HPC-systems [65]. However, environmental conditions can be more challenging in small edge clusters. For example, edge locations often have less efficient cooling systems than highly optimized hyperscale data centers; edge clusters also have less power redundancy, such as redundant battery and generator backups [50]. Moreover, CDN clusters employ a rigorous definition of server unavailability. When a server does not meet the performance requirement, it is deemed as unavailable by the load balancer. These types of unavailability are rarely reported by data centers and HPC systems. Unfortunately, the unavailability logs do not provide a causal breakdown of failure events.

Unavailability is mostly transient. Fig. 1c shows a CDF of the durations of unavailabilities. We observe that unavailabilities can last between 20 minutes and 24 hours with a median duration of 200 minutes. These short unavailabilities are mostly caused by performance degradation, such as unexpected server overload and software issues (e.g., application/kernel bugs or upgrades). Besides, we observe a long tail of unavailability durations, with around 16% exceeding 24 hours and 2% exceeding an entire week. These cases may be related to hardware issues. Qualitatively, our observations are similar to storage systems in the sense that unavailabilities are common and most unavailabilities are not permanent.

3.3 Mitigating unavailability is challenging

Upon detecting an unavailability, the load balancer removes the corresponding server from the consistent hash ring and reassigned their buckets to other servers [43]. We evaluate how a bucket’s object miss ratio is affected by unavailability using the video trace. Fig. 1d shows that the object miss ratio in a CDN cluster *without any redundancy* increases by more than 2 \times relative to no unavailability over the same time period. *This spike disproportionately affects a small group of content providers because of bucket-based routing* (§3.1).

The high latency resulting from cache misses can lead to SLA violations.

The state-of-the-art mitigation technique for server unavailability at large CDNs is *replicating* buckets across two servers². When one server becomes unavailable, requests are routed to the other server, likely to hold the object. Fig. 1d shows that replication reduces the intensity of the miss ratio spike. *In contrast to storage systems, where replication guarantees durability, in CDN clusters, servers perform cache evictions independently.* Objects that are admitted to two caches at the same time may be evicted at different times. This is particularly common if the two caches evict objects at very different rates, making replication ineffective. We next discuss why this case is more common than one might expect.

3.4 The need for write load balancing

We measure the read and write load balance across servers in a CDN cluster. To make the analysis independent from eviction decisions, we present the read and write rates based on compulsory misses from the web trace³. Table. 1 shows that the server with the highest read load serves 1.3 \times more traffic than the server with the lowest read load. The server with the highest write load writes around 2.5 \times more bytes than the server with the lowest write load.

Write load imbalance causes three problems. First, imbalance reduces the effectiveness of replication. A server with a 2.5 \times higher write rate also has a 2.5 \times higher eviction rate. So, a newly admitted object will traverse the cache with the highest write load 2.5 \times faster than the one with the least write load. Consequently, buckets mapped to these servers will have many objects for which only a single copy is cached in the cluster. We find that for 25% of objects, only a single copy exists in the cluster, which leads to the miss ratio spike observed during unavailabilities (Fig. 1d). Second, SSD write load imbalance often causes high tail latency. Specifically, high write rates frequently trigger garbage collection, which can delay subsequent reads by tens of milliseconds [11, 77, 78, 80]. These delays are significant enough to have been recognized as a problem by multiple CDN operators [61]. Third, the imbalance can lead to short SSD lifetimes due to concentrated writes on some SSDs, and thus higher replacement rates [9, 22], which increases CDN cost (§2).

4 C2DN System Design

C2DN’s design goals are to: (1) eliminate miss ratio spikes

²For operational flexibility, CDNs do not replicate servers as primary/backup. CDNs implement replication using additional virtual nodes for a bucket on the consistent hash ring [36, 47].

³Compulsory misses are cache admissions forced by objects not previously seen in the trace (underestimating the real miss and write rate). However, more compulsory misses only lead to more writes and evictions. Therefore, write rates are often proportional to compulsory misses.

caused by server unavailability, and (2) balance write loads across servers in the cluster. Erasure coding is a promising tool to improve availability under server unavailability. We first describe a naive implementation, called C2DN-NoRebal, based on a straightforward application of erasure coding (§4.1). C2DN-NoRebal fails to achieve the targeted goals, and we identify write and eviction imbalance as the key challenge. We then describe a new technique to overcome this challenge (§4.2) that exploits the unique aspects of the use of erasure coding in the context of CDNs.

4.1 Erasure coding and C2DN-NoRebal

Erasure coding is widely used in production storage systems for providing *high availability* with *low resource overhead* [32, 35, 48, 48, 54, 56]. Conceptually, erasure coding an object involves dividing the object into K *data chunks* and creating P parity chunks, which are mathematical functions of the data chunks. Such a scheme, called a (K, P) coding scheme, enables the system to decode the full object from any K out of the $K + P$ chunks. Thus, caching $K + P$ chunks on different servers provides tolerance to P server unavailabilities. As individual chunks are only a fraction $1/K$ of the original object’s size, coding reduces space overhead compared to replicating full objects⁴.

As CDNs use bucket-based routing (§2), coding needs to be applied at the level of buckets rather than objects. Specifically, the K data chunks of all the objects belonging to a bucket are grouped into K distinct *data buckets* respectively. Similarly, the corresponding P parity chunks are grouped into P distinct *parity buckets*. These buckets (data and parity) are each assigned to a distinct server in the cluster. Note that while the routing happens at the level of buckets, requests are still served at the level of objects. Hence we will use the term *buckets* in the context of assignment and *chunks* in the context of serving specific objects.

The application of erasure coding to CDNs is shown in Fig. 2a. To serve a user request, a server reads one chunk from the local cache and at least $K - 1$ chunks from other servers to reconstruct the requested object. To find the location of data and parity chunks, *C2DN-NoRebal* relies on a simple extension of bucket-based consistent hashing. The location of the first chunk is the server the bucket containing the object hashes to. Then, subsequent $K + P - 1$ chunks are read from the subsequent $K + P - 1$ servers on the consistent hash ring.

Owing to the reduced storage overhead, C2DN-NoRebal provides cost benefits by reducing the average byte miss ratio when compared to replication (as seen in our experiments in §6). However, C2DN-NoRebal *fails to eliminate the object miss ratio spike during unavailability* (§6). Specifically, we find that coded caches are even more sensitive to write load imbalance than replication. For replication, eviction rate imbal-

ance may cause the second (backup) copy to be evicted, which is required when a server becomes unavailable. Whereas for a coded cache, eviction rate imbalance could lead to any of the individual chunks being evicted, which leads to an effect we call *partial hits*: less than K chunks of the object are cached in the cluster, and this prohibits the reconstruction of the object. A partial hit only requires fetching the missing chunks, but incurs the same round-trip-time latency as a miss and thus does not provide a speedup. Further, partial hits become even more frequent during server unavailability, thus deeming C2DN-NoRebal less effective.

4.2 Parity rebalance and C2DN

Having identified write imbalance as a key challenge for erasure coding in CDNs, we next show how we exploit parities in overcoming these imbalances. Our main idea is to assign *parity buckets* to servers in a way that mitigates the write load imbalance caused by *data bucket* assignment.

Like the state-of-the-art in CDNs and C2DN-NoRebal, C2DN applies consistent hashing to assign the data buckets (Fig. 2b). We define a server’s data write load as the number of bytes written (i.e., admitted) to cache, counting only data buckets. We also define a bucket’s parity write load as the bytes written counting only parity buckets. Every server records data write load and each bucket’s parity write load since the cluster’s last unavailability event. After an unavailability event, parity buckets are reassigned by the load balancer using this information. The load balancer calculates an assignment of parity buckets to servers to balance write load. This assignment is a non-trivial calculation as not every assignment is feasible: parity chunks cannot be assigned to a server that holds a data chunk of the same object. In general, C2DN’s parity bucket assignment problem is NP-hard by reduction from the Generalized Assignment Problem [14].

C2DN’s parity bucket assignment algorithm. We obtain an approximate solution in polynomial time using a MaxFlow formulation (Fig. 2c). The solution provides us with feasible server assignments for each parity bucket. We empirically observe that by assigning the parity bucket to the least loaded server among the feasible servers, the write load on each server is well balanced. The inputs to the algorithm are:

1. parity write load of bucket n (s_n),
2. data write load on server i (l_i),
3. total write load on the cluster (W),
4. current assignment of data buckets to servers,
5. available servers in the cluster (\mathcal{A}).

The flow graph (Fig. 2c) is constructed using a source-node (S), parity-nodes corresponding to each parity bucket, server-nodes corresponding to each server in the cluster, and a sink-node (T). We add an edge from the source-node (S) to each parity-node n with a capacity equal to the bucket’s parity write load (s_n). We add edges from parity-nodes to the server-nodes if the corresponding parity bucket can be placed on that

⁴The space overhead of an (K, P) coding scheme is $\frac{K+P}{K}$. For example, for $K = 3, P = 1$, space overhead is $1.33 \times$ as opposed to $2 \times$ in two-replication.

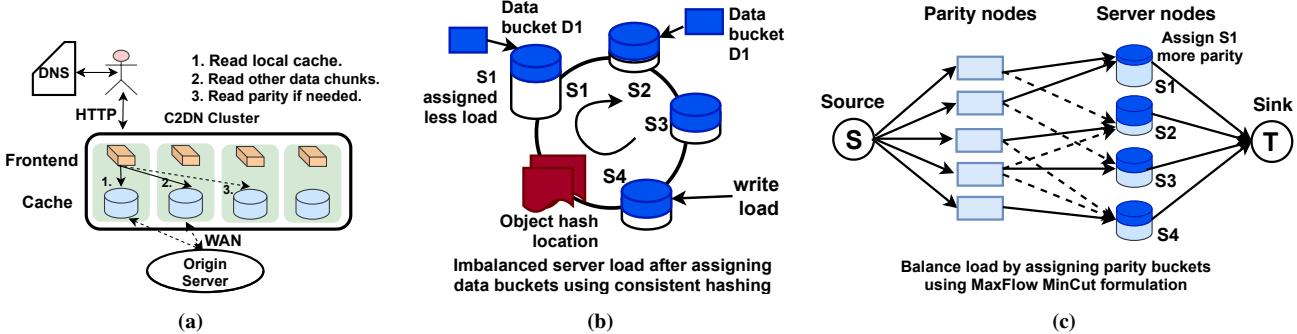


Figure 2: a) **Architecture of C2DN**; b) **C2DN bucket assignment** C2DN assigns data buckets using consistent hashing, which guarantees a consistent mapping across unavailabilities, but causes load imbalance; c) **Parity rebalance**. Write load imbalance is mitigated by assigning parity buckets to balance the load using a MaxFlow formulation

server, i.e., the data chunks of the bucket are not assigned to the server. The capacity of these edges is again the bucket’s parity write load (s_n). Finally, we add edges from server-nodes to the sink-node (T) with a capacity equal to the server’s remaining write load budget, which is $\max(\left\lceil \frac{W}{|\mathcal{A}|} \right\rceil - l_i, 0)$.

After solving MaxFlow(S, T), C2DN iterates over parity buckets. Each parity bucket is assigned to the least loaded server with a positive flow from the parity-node to the server-nodes. This leads to a well-balanced assignment. The assignment is also feasible as no positive flow exists between a parity bucket and the servers holding this bucket’s data chunks.

The parity rebalance algorithm is described in more detail via a pseudo-code in Appendix 10.3.

Extension to heterogeneous servers. We incorporate heterogeneous servers by setting the capacity of the edge in the graph between server-nodes to sink-node (T) proportional to the size of the server.

4.3 C2DN resolves partial hits

Having shown how to balance write loads across servers within the cluster, we show that this is sufficient to solve the issue of partial hits. Specifically, we find that the probability of a partial hit diminishes for large caches.

We formulate our proof under the simplifying assumptions of the independent reference model (IRM⁵), which is used widely in caching analysis [5, 10, 23]. While our proof can be extended to a range of eviction policies [44], we assume the Least-Recently-Used (LRU) policy for simplicity. We empirically observe that FIFO, which is used in open-source caches such as Apache Trafficserver [7] and our empirical evaluation in §6, behaves similarly to LRU.

We remark that we *do not require explicit coordination of individual eviction decisions among the caches*. Our theorem states that under IRM, in C2DN, if one chunk of an object is present in a cache, then the other chunks are almost surely

⁵In the IRM, an object i ’s requests arrive according to a Poisson process with a rate λ_i , independent of the other objects’ requests. With recent theoretical advances [34], our proof can be extended to not assume the IRM.

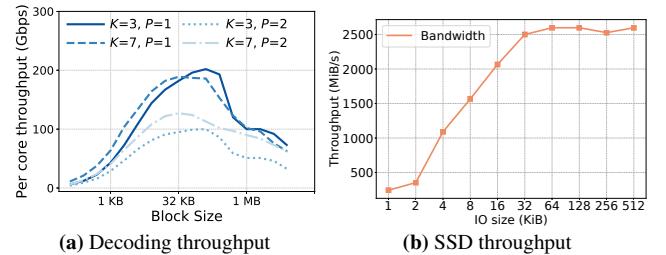


Figure 3: Microbenchmarks. a) With vector instruction in modern CPU, decoding is very efficient with high throughput, the sub-chunk size to achieve maximum throughput across configurations is around 32-64 KB. b) Modern SSD achieves maximum throughput with I/O size larger than 32 KB.

present in the other caches.

Theorem 1. Under IRM and LRU, in C2DN, for an object with chunks x_1, \dots, x_n , for any $1 \leq i, j \leq n$, and as the cache size grows large

$$P[\text{chunk } x_i \text{ is in cache} \mid \text{chunk } x_j \text{ is in cache}] \rightarrow 1 \quad (1)$$

Our proof uses the fact that balanced write loads lead to equal characteristic times [10, 23, 26, 60], which is the time it takes for a newly requested chunk to get evicted from each server’s LRU list. Since data and parity chunks of an object are requested simultaneously and the characteristic time is the same, the chunks are also evicted simultaneously, and partial hits become rare. Details can be found in Appendix 10.2.

5 C2DN Implementation

In addition to design goals (1) and (2), C2DN’s implementation seeks to (3) minimize storage/ latency/ CPU overheads and (4) remain compatible with existing systems to facilitate deployment. This entails subtle implementation challenges.

Enabling transparent coding. A key architectural question is which system component encodes and decodes objects into/from data and parity chunks. A natural choice might be to encode objects at origin servers. However, this would require changes to thousands of heterogeneous origin software stacks — a barrier to deployment. Additionally, encoding at the origin

would increase origin traffic as each cache miss needs to fetch both data and parity chunks, e.g., with $K = 3, P = 1$ the origin traffic would increase by 33%. Thus, C2DN fetches uncoded objects from origins and encodes chunks within the CDN cluster. Additionally, any decoding operation is also performed within the cluster for transparency on the client side.

Selective erasure coding. While encoding and decoding are fast due to broad CPU support for vector operations, the overhead of fetching becomes significant for small objects. As the majority of requests are for small objects (§3.1), we can reduce processing overheads by using replication for small objects. C2DN applies coding to large objects, which account for most of the production cluster’s cache space (§3.1). Of course, with selective coding, we now need to count uncoded objects as part of the data write load in §4.2.

To decide the size threshold of coding, we perform two microbenchmarks studying how coding block size affects coding throughput and SSD bandwidth. Fig. 3a shows that even on a five-year-old Skylake Xeon, decoding is very efficient with per-core throughput over 200 Gbps (data fits in CPU cache) at a block size of 32 KB. This benchmark result suggests that decoding will not be a bottleneck at a reasonable block size (e.g., 32 KB) compared to NIC bandwidth. Fig. 3b shows the relationship between SSD bandwidth and I/O size (setup as in §6). We again find that a block size of 32-64 KB achieves the peak SSD bandwidth. Based on these results, C2DN codes object larger than 128KB so that each chunk is at least 42KB for a (3, 1) coding scheme.

This hybrid approach enables load balancing and space efficiency with no overhead for most requests. One might ask why C2DN relies on replication for small objects after §2 showed that replication continues to suffer from miss ratio spikes. We find that erasure coding large objects is sufficient to balance eviction rates (using C2DN’s parity rebalance), making replication effective for small objects.

Parity rebalance and parity look up. As described in § 4.2, C2DN formulates the parity bucket assignment problem as a Max Flow problem. We solve the problem using Google-OR [53], which implements the push-relabel algorithm [18]. The time complexity of this algorithm is $O(n_{node}^2 * \sqrt{n_{edge}})$, where n_{node} is the number of nodes (#buckets + #servers) and n_{edge} is the number of edges (\approx #buckets \times #servers). In production systems, #buckets is in the range of 100s for a 10-server cluster. Thus, the time complexity simplifies to $O(\#buckets^3)$. Empirically, we observe low run times as well, for e.g., for 100 buckets and 10 servers, C2DN’s parity bucket assignment runs within 50 μ s. Also, note that the parity bucket mapping is calculated in the background (off the critical path) and only when there is an unavailability event. From our analysis, we observe around 5.6 unavailability events on an average day.

Support for large file serving, HTTP streaming, and byte-range requests. To minimize latency, CDNs stream large ob-

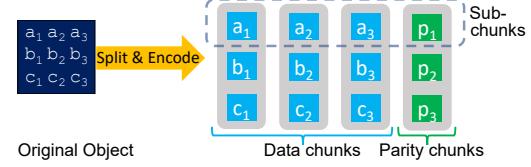


Figure 4: Support for HTTP streaming. C2DN efficiently supports HTTP streaming and byte-range requests by splitting large files into sub-chunks and performs coding on sub-chunks level.

System	Replication(CDN)	C2DN	C2DN reduction
Object miss ratio	0.242	0.227	6.4%
Byte miss ratio	0.118	0.105	11%

Table 2: Object and byte miss ratio from prototype

jects to clients. We achieve compatibility with streaming by subdividing data and parity chunks (for very large objects) into smaller parts which we call sub-chunks. C2DN’s encoding and decoding work on the sub-chunk level as shown in Fig. 4. We implement streaming by serving sub-chunks as they become available. For byte-range requests, C2DN fetches the sub-chunks overlapping with the requested byte-range.

Delayed fetch of parity sub-chunks. C2DN can serve a request with any K sub-chunks (out of $K + P$). Because serving with data sub-chunks requires no decoding, C2DN first fetches all K data sub-chunks. C2DN only fetches parity sub-chunks after a heuristic wait period to overcome stragglers. We record the time until the first data sub-chunk is returned. If, after an additional 20% wait time, fewer than K data sub-chunks have arrived, C2DN fetches parity sub-chunks.

Hot object cache (HOC). To facilitate serving hot objects, C2DN caches decoded sub-chunks in DRAM so that if an object is popular, it will be served directly and efficiently from DRAM, thus avoiding fetching and possible decoding.

Metadata lookups. In the case of a HOC miss, C2DN needs to know if the object was encoded or replicated. Storage systems can rely on external metadata for this case, which is not available in CDNs. Thus, C2DN stores metadata with each cached object, indicating whether the object is coded or not. On a HOC miss, C2DN first looks up the object in its *local SSD cache*. If the metadata indicates a coded object, C2DN fetches chunks from other caching servers within the cluster. In the case of a local cache miss, C2DN retrieves the object from other CDN clusters or the origin servers, then C2DN serves the object to the end-user, stores it locally, and encodes or replicates based on the object size.

6 Evaluation

We build C2DN on top of Apache Trafficserver and evaluate it via a series of experiments on Amazon EC2. To study a more comprehensive parameter range, we use simulations. The source code of our prototype and simulator is released at <https://github.com/TheSys-lab/C2DN>.

The highlights of our evaluation are: (1) C2DN eliminates miss ratio spikes after unavailabilities. Additionally, C2DN re-

duces byte miss ratio by 11%, enabling significant bandwidth cost savings at scale. (2) C2DN reduces write load imbalance by 99%. (3) C2DN achieves the same latency, lower average SSD write rates with only a 14% increase in CPU utilization.

6.1 Experimental methodology and setup

Traces. We evaluate C2DN using the two production traces described in §3. In the following sections, we focus on the video trace and present results for the web trace in §6.7.

Prototype evaluation setup. We emulate a CDN’s geographic distribution by placing sets of clients, a 10-server CDN cluster, and an origin data center in different AWS regions. CDN servers use i3en.6xlarge VMs with 80 GB in-memory cache and 10 TB disk cache. To reduce WAN monetary bandwidth costs of the experiments, we measure latency via spatial sampling [72, 73] for 2% of requests. The remaining requests are generated in the same region.

Unless specified otherwise, we use Reed-Solomon codes ($K = 3, P = 1$). We only code objects larger than 128 KB (§4). The prototype experiments use four days of requests to warm up caches. Measurements are then taken for three days of requests. This corresponds to replaying 1.18 PB of traffic in total from local and remote clients in each prototype experiment.

Simulation setup. We implement a request-level cluster simulator. While the simulator does not capture system overheads, it is useful in comparing various schemes for the full duration of the trace and for various cache sizes (which are prohibitively expensive to perform using prototype experiments.) Simulations use 18-day long traces (compared to 7 days with the prototype). Unless otherwise stated, the simulator uses the same configuration as the prototype.

Baselines. We compare C2DN to three baselines. (1) **No-replication** does not provide fault tolerance and incurs no space overhead. (2) **Replication (CDN)** replicates each object with two replicas. We use the (CDN) suffix as this is most similar to the approach deployed today. (3) **C2DN-NoRebal** a C2DN variant based on consistent hashing without parity rebalance. In addition to C2DN, which uses one parity chunk and tolerates one unavailability, we have also evaluated C2DN-n5k3 and C2DN-n6k3, which uses two and three parity chunks, and can tolerate two and three unavailabilities, respectively.

6.2 Miss ratio without unavailability

We evaluate miss ratios of the competing systems under normal operation, i.e., *without unavailability*. Table. 2 shows the object miss ratio and byte miss ratio of Replication (CDN) and C2DN obtained from the prototype experiments. We observe that C2DN reduces object miss ratio by 6.4% and byte miss ratio by 11.0%. These improvements are direct results of the reduced storage overhead in C2DN. At a large scale, these improvements lead to significant bandwidth savings.

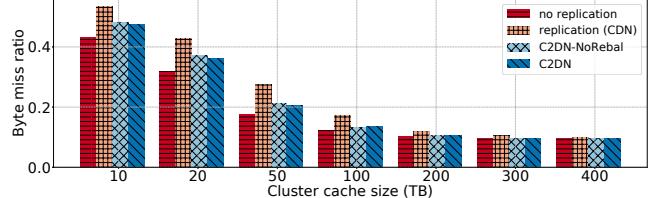


Figure 5: Byte miss ratio of the four systems.

To understand the sensitivity of byte miss ratio improvements to cache size, we show simulation results in Fig. 5. For smaller cache sizes, C2DN improves byte miss ratios by up to 20%. Benefits diminish for cache sizes above 200 TB ($5 \times$ production cache size). For object miss ratios, the effect is qualitatively similar (Fig. 12 in the appendix). Overall, the reduction in miss ratio bridges the efficiency gap between No-replication and Replication (CDN) and reduces the overhead of providing redundancy in CDN edge clusters.

We also observe that C2DN improves miss ratios compared to C2DN-NoRebal because C2DN balances the write loads (eviction rates) across servers and reduces the probability of partial hits. However, this effect is small, suggesting that most of C2DN’s miss ratio reduction comes from reduced storage overhead. The advantage of C2DN over C2DN-NoRebal will become clear in the following section, where we find that C2DN-NoRebal does not provide effective fault tolerance.

6.3 Miss ratio under unavailability

We now consider unavailabilities and evaluate the object miss ratio as the primary performance metric affecting latency and speedup. A first experiment introduces single unavailability after warming up the cache. We then measure the relative object miss ratio change: $\frac{mr(un) - mr(av)}{mr(av)}$ for each 5 minute time interval, where $mr(un)$ and $mr(av)$ stand for miss ratio with unavailability and without unavailability, respectively. A second experiment considers two simultaneous unavailabilities.

Fig. 6a show the relative object miss ratio increase where the single unavailability event occurs 100 minutes after warmup. As expected, No-replication does not provide fault tolerance, leading to a large (2.2 \times as seen in 1d) miss ratio spike. Replication (CDN) and C2DN-NoRebal have similar performance with 25% miss ratio spikes.

The miss ratio of C2DN is not affected for several hours after the unavailability event. This is because C2DN with one parity chunk can tolerate one unavailability effectively. In the long term, miss ratios for all systems increase as the cluster’s total capacity is reduced. For C2DN, the increase in the miss ratio becomes visible only after around 300 minutes past unavailability. During unavailability, data that should be written to the unavailable servers are written to the other available servers. The extra writes take a long time to impact the miss ratio of clusters with a large cache size. We remark that the exact length of such no performance degradation is not fixed and is dependent on the trace.

The reason for the miss ratio spike in Replication (CDN)

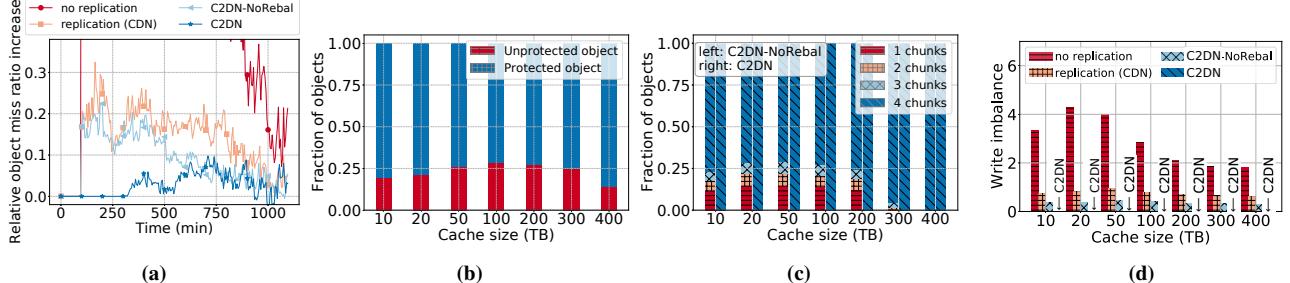


Figure 6: a) Replicated CDN mitigates unavailability, but still has a spike after unavailability. C2DN mitigates the unavailability spike. b) Servers in the Replicated CDN evict objects independently, and due to write imbalance this leads to unprotected objects. c) Naive coding has similar problems as replication; C2DN solves this problem by parity rebalance. d) Write load imbalance for different systems across various cache sizes.

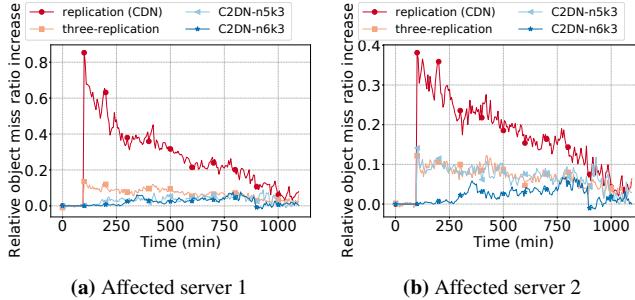


Figure 7: With two simultaneous unavailabilities, two replication (CDN) shows a big spike when the unavailability happens. Three replication and C2DN-n5k3 still show a small spike due to evicted replica/chunk. C2DN-n6k3 completely eliminates the spike.

and C2DN-NoRebal—despite using redundancy—is the severe write and eviction rate imbalance in these systems (§3.3 and 3.4). This imbalance leads to *unprotected* objects: an object is unprotected if only a single copy is cached in the cluster. For C2DN-NoRebal, unprotected objects are objects with fewer than $K + 1$ chunks cached in the cluster.

Fig. 6b shows the fraction of (un)protected objects in Replication (CDN). We observe that more than 25% of objects can be unprotected. The fraction of unprotected objects initially increases with cache size and then decreases. This pattern is because only highly popular objects are cached when the cache size is small, and hence the chance of having both replicas is higher. On the other hand, replicas are less likely to be evicted when the cache size is very large. Fig. 6c shows the fraction of unprotected objects in C2DN-NoRebal and C2DN. Since $K = 3$ and $P = 1$, objects with fewer than 4 chunks are unprotected. For caches smaller than 300 TB, up to 24% of the objects in C2DN-NoRebal are unprotected. In contrast, C2DN protects nearly 100% of cached objects across all cache sizes and effectively eliminates miss ratio spikes.

So far, we have only focused on one unavailability. When a CDN operator seeks to tolerate more than one unavailability, C2DN’s advantage over replication increases as the space requirements for erasure coding scale significantly better. As an empirical data point, we consider two unavailabilities and compare two-replication and three-replication with C2DN-n5k3 and C2DN-n6k3. C2DN-n5k3 (C2DN-n6k3) uses two

System/server load	Max	Min	Mean	Max/min
CDN write (TB)	16.83	9.26	13.48	1.82
C2DN write (TB)	8.44	8.40	8.42	1.00

Table 3: Write load on servers in Replication (CDN) and C2DN.

(three) parity chunks with 66% (100%) storage overhead and can tolerate two (three) unavailabilities. In contrast, two-replication and three-replication tolerate one and two unavailabilities with 100% and 200% storage overhead, respectively.

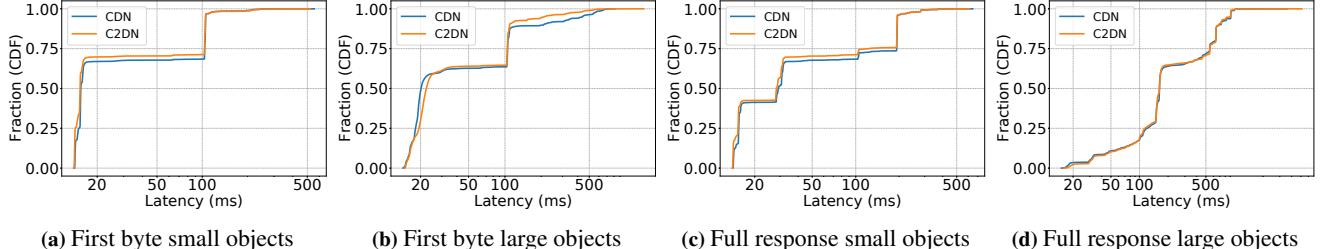
Fig. 7 shows that compared to two-replication, C2DN-n5k3 and three-replication significantly reduce the miss ratio spike from over 80% to less than 20%. Furthermore, the miss ratio spike disappears entirely with C2DN-n6k3, which has the same storage overhead as two-replication.

6.4 Write (Read) load balancing

We quantify how well systems balance write load across servers. Balancing writes is the key to mitigating miss ratio spikes and helps control SSD tail latency and endurance.

Table. 3 shows bytes written per server in our prototype experiments. The busiest server in Replication (CDN) writes 16.8 TB compared to 8.4 TB for the busiest server in C2DN. With half the write rate, C2DN may double SSD lifetime and reduce tail latency by up to an order of magnitude [80]. The write imbalance in Replication (CDN) between peak and minimum write rate is $1.82 \times$. In contrast, the write imbalance in C2DN is less than $1.005 \times$. We also observe that C2DN reduces read imbalance from $1.69 \times$ for Replication (CDN) to $1.34 \times$. The read imbalance in C2DN remains as parity rebalancing (§4.2) focuses exclusively on write rate.

We further explore the effects of load balancing across various cache sizes using simulations. If M is the write (read) load on the server with maximum write (read) load and m is the minimum write (read) load across the servers, then write (read) load imbalance = $\frac{M-m}{m}$. Fig. 6d shows that C2DN eliminates write imbalance for all cache sizes. When averaged across cache sizes, C2DN reduces the **write** load imbalance by 99.9% compared to No-replication, 99.8% compared to Replication (CDN), and 99.5% compared to C2DN-NoRebal. C2DN also reduces the **read** load imbalance: by



(a) First byte small objects

(b) First byte large objects

(c) Full response small objects

(d) Full response large objects

Figure 8: First-byte and full response latency of serving small and large objects in CDN and C2DN.

93.9% compared to No-replication, 78.9% compared to Replication (CDN), and 70.5% compared to C2DN-NoRebal on an average across the different cache sizes.

6.5 Latency

We quantify potential latency overheads by measuring the time-to-first-byte (TTFB) and content download time (CDT) of our prototype implementations of C2DN and Replication (CDN). In each case, we separately measure the latency distribution for objects below the 128KB coding threshold (“small” objects) and for objects above the threshold (“large” objects). Fig. 8a and Fig. 8b show the cumulative distributions of TTFB for small and large objects, respectively. For small objects, we find that the TTFB distributions for C2DN and Replication (CDN) are similar, as expected: C2DN does not code these objects. C2DN slightly improves the TTFB distribution (shifting to the left) due to its lower object miss ratio. For large objects, we find about a 1 ms overhead in TTFB at low percentiles (25th-60th percentile). The slight increase is for cache hits due to fetching the first sub-chunk from K servers before serving the object.

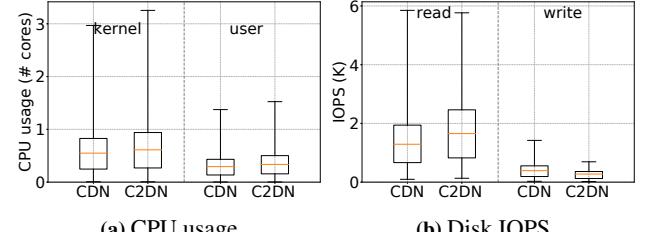
We now consider CDT. In practice, this metric is more relevant for large objects than the TTFB. Fig. 8c and Fig. 8d show a cumulative distribution of the content download time for small and large objects, respectively. Again we find that small objects behave similarly in Replication (CDN) and C2DN, with slightly better latency for C2DN due to lower object miss ratio. For large objects, C2DN and Replication (CDN) have a similar CDT. The overheads of fetching chunks are hidden by our streaming implementation based on sub-chunks (§5).

We remark that C2DN improves the tail latency in all cases (barely visible in the CDFs). For example, C2DN reduces the P90 TTFB by up to $3\times$ compared to Replication (CDN). We attribute this to a lower miss ratio and the mitigation of stragglers using parity chunks to serve requests. This is as expected based on prior work on using coding to reduce tail latency [55].

6.6 Overhead assessment

We quantify the resource overheads of our C2DN prototype.

CPU usage. Fig. 9a measures CPU utilization in fractional CPU cores for userspace and kernel tasks, respectively. C2DN generally leads to higher CPU usage. The userspace CPU usage is higher due to the encoding and decoding of objects, and



(a) CPU usage

(b) Disk IOPS

Figure 9: Resource usage. C2DN uses slightly more CPU resources and slightly more read disk IOPS than CDN, however, C2DN reduces write disk IOPS, especially at peak.

the kernel CPU usage is higher due to additional network and disk I/Os. Overall, CPU usage increases by 14% on average with a similar increase in kernel and userspace CPU usage.

The increase in the CPU overhead is small as C2DN performs the encoding and decoding operation only on a fraction of requests. For the current coding size threshold of 128 KB, the number of requests served with coded objects is around 50%, while the number of bytes served using coded objects is close to 90%. Also, recall that most requests for coded objects do not need to be decoded as the object is recreated by concatenating data chunks in the output buffer. Decoding only happens in the case of stragglers and partial hits. In fact, only 6% of requests require decoding in our experiments. These cases happen primarily due to the straggler problem (individual slow servers); actual data chunk misses (partial hits) occur for less than 0.6% of requests. A future version of C2DN may further reduce CPU overheads by using kernel-bypass networking or increasing the object size threshold for coding. Increasing the threshold can happen with minimal side effects, as we show in the next section.

Disk usage. Fig. 9b compares disk IOPS of Replication (CDN) and C2DN for reads and writes. For reads, we observe that C2DN uses 23% more IOPS in the mean and less at the tail. Read-IOPS increase by 2% at the P99 and decrease by 11% at the P99.9 (we calculate this percentile across time and servers). For writes, C2DN uses 24% fewer writes IOPS in the mean. The tail write IOPS decreases by 46% at the P99 and 50% at the P99.9. The read IOPS increases because C2DN fetches at least $K = 3$ chunks to serve an object if coded. However, due to 1) most of the requests being for small uncoded objects, 2) the presence of DRAM hot object cache, the increase in reading IOPS is much smaller than $3\times$. While mean read IOPS increase, the peak read IOPS is similar or lower in C2DN. We attribute this to better load balancing

in C2DN. Write IOPS in C2DN is significantly reduced when compared to Replication (CDN). C2DN has a lower storage overhead than Replication (CDN) and thus writes less to disk. In addition, the improvement in the miss ratio that C2DN provides further reduces the number of write operations. Besides, C2DN also improves the tail write IOPS, which is due to a better load balancing strategy of erasure coding and parity rebalance.

Intra-cluster network usage. C2DN uses network bandwidth within the cluster, about 0.9 Gbps in the mean and 2.3 Gbps at the P95. In conversations with CDN operators, this internal bandwidth usage is feasible for production clusters, as these links generally show little usage. For example, production CDN clusters use dedicated 10-Gbps-NICs for communication within the cluster.

6.7 Sensitivity analysis

We discuss the sensitivity of C2DN to its parameters.

Coding size threshold. The size threshold for coding impacts the performance in multiple ways. By reducing the size threshold, C2DN encodes more objects, improving cache space usage and load balance across cluster servers. At the same time, it leads to more CPU and I/Os (due to coding and fetching) and increases the latency for small objects. The size distribution in Fig. 1a shows that small objects contribute a small fraction of cache space usage. Thus, the potential benefit of coding diminishes as we decrease the size threshold for coding. At the same time, C2DN would use more cluster resources. We observe that reducing the size threshold to below 128 KB does not significantly benefit the object and byte miss ratio. Increasing the size threshold to over 8 MB increases the byte miss ratio by 2.79% and the write load imbalance by 258%. We believe that 128 KB is a good tradeoff for our production traces. Fig. 13 in the appendix shows our results.

Coding parameter K . Most of this section assumed C2DN configured with $K = 3$. We explore the impact of parameter K and P on miss ratio and write load balancing. We find that increasing K and keeping P constant reduces miss ratios for C2DN but increases miss ratios for C2DN-NoRebal. When adding chunks, the probability of getting partial hits increases for C2DN-NoRebal due to unbalanced eviction rates between servers. Because C2DN uses parity rebalance to achieve similar eviction rates between servers, the miss ratio decreases with increasing K due to lower storage overhead. While the impact of coding parameters has different impacts on miss ratios for C2DN-NoRebal and C2DN, the impact on load balancing is similar, as K increases, because an object is broken into more (and smaller) chunks, both the read and write load imbalance in C2DN-NoRebal and C2DN reduce. Fig. 14 in the appendix shows our results.

Different workloads. Throughout this section, we have used the video trace. We repeated our evaluation for the week-long web trace (§3). Compared to the video trace, the web trace has a significantly smaller working set. The video trace has a

compulsory byte miss ratio of 0.1 and a compulsory object miss ratio of 0.21. In the web trace, the compulsory miss ratio is 0.06 for both byte and object miss ratios. In addition, compared to the video trace, the web trace has a more diverse object size range, as shown in Fig. 1a. Less than 10% of large objects (possibly large software) contribute to more than 90% of the cache space usage. Therefore, the fraction of requests that require coding is significantly smaller.

In prototype experiments with the web trace, only 3% of all requests are served coded. However, these 3% of requests account for 80% of served traffic. As a comparison, in the video trace, the prototype serves about 50% of requests from coded objects (with only 6% requiring decoding). Consequently, coding overheads on the web trace are negligible. In terms of the miss ratio, we observe a 10% reduction in object miss ratio and a 6% reduction in byte miss ratio. The write imbalance for Replication (CDN) is $1.72\times$, which is reduced to $1.03\times$ in C2DN. The read imbalance for Replication (CDN) is $4.8\times$, which is reduced to $2.5\times$ in C2DN.

Different eviction algorithms. Throughout this section, we have used FIFO as the eviction algorithm for the cache. FIFO provides stable performance on SSDs and extends the lifetime of an SSD by minimizing device write amplification [9, 22, 70]. Many open source caches such as Apache Trafficserver [7] and Varnish [71] use FIFO. To understand the impact of the eviction algorithm, we evaluate the Least-recently-used algorithm (LRU) using simulation. We observe a slight reduction in both object and byte miss ratios for all systems. All other results are qualitatively and quantitatively the same. Appendix 10.4, Fig. 16 and Fig. 17 show these results.

Variants of replication. Besides two-replication for all objects, CDNs have explored systems that replicate based on popularity. Specifically, only popular objects are replicated on two servers to reduce space overheads. As might be expected from our findings that write imbalance matters, popularity-based replication does not provide good fault tolerance. In simulation experiments, we observe object miss ratio spikes by 82%. Interestingly, we also observe that popularity-based replication leads to an even worse load imbalance than Replication (CDN), which explains the high miss ratio spike.

7 Discussion

DNS vs anycast-based CDN request routing. Different CDNs use different global load balancing architectures [33]. Akamai is well known for its DNS architecture [64]. Limelight [33], Wikipedia [58], and Cloudflare rely on anycast. While these designs have different performance implications, both rely on algorithms like consistent hashing. In DNS-based systems, consistent hashing is applied by the cluster-local load balancer to return the IP of the server responsible for the shard. Anycast-based systems typically route requests to any server in a cluster, and the server uses consistent hashing to identify another server that likely stores the object. Server

unavailability, storage overheads of redundancy, and write imbalance are important problems in all CDN designs. While the cluster-local load balancer in our prototype relies on DNS, the principle design components of C2DN can be equally applied in anycast systems. We also expect that C2DN’s benefits will transfer with similar quantitative improvements.

Larger clusters and multiple unavailabilities. In clusters of large size, multiple concurrent unavailabilities are not uncommon. As evaluated in §6, we find that C2DN is more effective in this setting as erasure coding is more efficient at tolerating multiple unavailabilities than replication. For large clusters, server unavailabilities become more common. We thus recommend either using a coding scheme with more parity chunks or handling the cluster as multiple smaller clusters.

8 Related work

While there is extensive work on caching, coding, load balancing, and flash caching, our work is uniquely positioned at the intersection of these areas. We discuss work by area.

Erasure coding in storage systems. Prior work has characterized the cost advantage offered by coding over replication in achieving data durability in distributed storage systems [75, 85]. Erasure codes are deployed in RAID [52], network-attached-storage [4], peer-to-peer storage systems [37, 40, 57, 79], in-memory key-value store [16, 17, 84], and distributed storage systems [32, 48, 56, 76]. Coding for CDNs differs due to the unique interplay of coding and caching and the two-sided transparency requirement (§4). Additionally, CDNs employ coding for different reasons (performance) than storage systems (durability), which magnifies overhead concerns.

Caching for coded file systems. Several recent works have explored augmenting erasure-coded storage systems with a cache to reduce latency [3, 29, 41, 55]. Aggarwal et al. [3] proposed augmenting erasure-coded disk-based storage systems with an in-memory cache at the proxy or the client-side that cache encoded chunks. Halalai et al. [29] propose augmenting geo-distributed erasure-coded storage systems by caching a fraction of the coded chunks in different geo-locations to alleviate the latency impact of fetching chunks from remote geo-locations. EC-Cache [55] employs erasure coding in the in-memory layer of a tiered distributed file system such as Alluxio (formerly [39]). Although EC-Cache is technically a cache, there is no interaction between coding and caching in EC-Cache since it operates in scenarios where the entire working set fits in memory, i.e., no evictions are considered. In contrast, C2DN focuses on CDN clusters with working sets in the hundreds of TB and starkly different tradeoffs, workload characteristics, and challenges as compared to file systems. In the area of cooperative caching [6, 30, 62], nodes synchronize caching decision via explicit communication. In contrast, C2DN proves that explicit communication is not required to synchronize the eviction of the K chunks, which significantly decreases overheads.

Chunking and caching. Prior work has explored the chal-

lenges of serving large files over HTTP, e.g., CoDeeN [74]. Similar to C2DN, CoDeeN breaks a large file into smaller chunks. A chunk cache miss does not require transferring the whole large file from the origin. In contrast to CoDeeN, C2DN addresses unavailability tolerance, which is not provided by chunking alone.

Load balancing. Load balancing and sharding are well-studied topics [1, 2, 12, 13, 21, 27, 28]. To reduce the load imbalance, John et al. study the power of two choices that reduces the imbalance [12]. In addition, to serve skewed workloads, Fan et al. [24] study the effect of using a small and fast popularity-based cache to reduce load imbalance between different caches in a large backend pool. Yu-ju et al. [31] designed SPORE to use a self-adapting, popularity-based replication to mitigate load imbalance. Rashmi et al. [55] used erasure coding to reduce read load imbalance for large object in-memory cache. In summary, prior work on load balancing focuses on *read* load balancing, with little attention paid to *write* load balancing.

Load imbalance in consistent hashing can be solved with additional lookups via probing [47]. Unfortunately, these lookups are costly in CDNs (particularly for DNS-based systems). Additionally, this approach cannot be applied for erasure-coded caches due to the constraint that parity chunks should not be colocated with data chunks. In contrast to using load balancing to achieve a similar SSD replacement time, Mahesh et al. [8] used parity placement to achieve differential SSD ages so that SSDs of a disk array fail at different times.

Flash cache endurance. Flash caching is an active and challenging research area. A line of work [38, 45, 51, 63, 66, 67, 70] shows how eviction policies can be efficiently implemented on flash. Flashield [22] proposes to extend SSD lifetime via smart admission policies. All these systems focus on a single SSD. Our work focuses on wear-leveling across servers in a cluster, which significantly extends the lifetime of a cluster.

9 Conclusion

We re-architected the cluster of a CDN by introducing a hybrid redundancy scheme using erasure codes and replication, along with a novel approach for parity placement. We showed that our approach reduces the miss ratio and eliminates the miss ratio spikes caused by server unavailability. Further, our approach is more space-efficient than replication and is more attractive as CDN traffic and content footprint scale rapidly with Internet usage. Finally, our approach also reduces the write load imbalance by optimally placing the parities to reduce the lifetime of SSDs. We believe that C2DN is attractive for deployment in a production CDN since it integrates well with production CDN components.

Acknowledgements We thank our shepherd Angela Demke Brown and the anonymous reviewers for their valuable feedback. This work was supported in part by Facebook Fellowship, NSF grants CNS 1901410, CNS 1956271, CNS 1763617, and a AWS grant.

References

- [1] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, Apr. 2010. USENIX Association.
- [2] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khezani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, et al. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, 2016.
- [3] V. Aggarwal, Y.-F. R. Chen, T. Lan, and Y. Xiang. Sprout: A functional caching approach to minimize service latency in erasure-coded storage. In *ICDCS*, 2016.
- [4] M. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *DSN*, 2005.
- [5] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *Journal of the ACM (JACM)*, 18(1):80–93, 1971.
- [6] S. Annareddy, M. J. Freedman, and D. Mazieres. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-VOLUME 2*, pages 129–142. USENIX Association, 2005.
- [7] Apache. Traffic Server, 2019. Available at <https://trafficserver.apache.org/>, accessed 09/18/19.
- [8] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi. Differential raid: Rethinking raid for ssd reliability. *ACM Transactions on Storage (TOS)*, 6(2):1–22, 2010.
- [9] B. Berg, D. S. Berger, S. McAllister, I. Grososf, S. Guneskar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, et al. The cachelib caching engine: Design and experiences at scale. In *USENIX OSDI*, pages 753–768, 2020.
- [10] D. S. Berger, R. Sitaraman, and M. Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a cdn. In *USENIX NSDI*, pages 483–498, March 2017.
- [11] M. Bjørling, J. Gonzalez, and P. Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, Feb. 2017. USENIX Association.
- [12] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *International Workshop on Peer-to-Peer Systems*, pages 80–87. Springer, 2003.
- [13] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC ’11, New York, NY, USA, 2011. Association for Computing Machinery.
- [14] D. G. Cattrysse and L. N. Van Wassenhove. A survey of algorithms for the generalized assignment problem. *European journal of operational research*, 60(3):260–272, 1992.
- [15] F. Chen, R. K. Sitaraman, and M. Torres. End-user mapping: Next generation request routing for content delivery. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 167–181. ACM, 2015.
- [16] H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang. Efficient and available in-memory kv-store with hybrid erasure coding and replication. *ACM Transactions on Storage (TOS)*, 13(3):1–30, 2017.
- [17] L. Cheng, Y. Hu, and P. P. C. Lee. Coupling decentralized key-value stores with erasure coding. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC ’19, page 377–389, New York, NY, USA, 2019. Association for Computing Machinery.
- [18] B. V. Cherkassky and A. V. Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.
- [19] CISCO. Global IP traffic forecast: The zettabyte era—trends and analysis, June 2017. Available at <https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.pdf>, accessed 24/09/17.
- [20] J. Dilley, B. M. Maggs, J. Parikh, H. Prokop, R. K. Sitaraman, and W. E. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.
- [21] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.
- [22] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked*

- Systems Design and Implementation (NSDI 19)*, pages 65–78, Boston, MA, Feb. 2019. USENIX Association.
- [23] R. Fagin. Asymptotic miss ratios over independent references. *Journal of Computer and System Sciences*, 14(2):222–250, 1977.
- [24] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–12, 2011.
- [25] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.
- [26] C. Fricker, P. Robert, and J. Roberts. A versatile and accurate approximation for LRU cache performance. In *ITC*, page 8, 2012.
- [27] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.
- [28] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. Basil: Automated io load balancing across storage devices. In *Fast*, volume 10, pages 13–13, 2010.
- [29] R. Halalai, P. Felber, A.-M. Kermarrec, and F. Taiani. Agar: A caching system for erasure-coded data. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 23–33. IEEE, 2017.
- [30] V. Holmedahl, B. Smith, and T. Yang. Cooperative caching of dynamic content on a distributed web server. In *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No. 98TB100244)*, pages 243–250. IEEE, 1998.
- [31] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–17, 2013.
- [32] C. Huang, H. Simitci, Y. Xu, A. Oqus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX ATC*, 2012.
- [33] C. Huang, A. Wang, J. Li, and K. W. Ross. Measuring and evaluating large-scale cdns. In *ACM IMC*, volume 8, pages 15–29, 2008.
- [34] B. Jiang, P. Nain, and D. Towsley. On the convergence of the ttl approximation for an lru cache under independent stationary request processes. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(4), Sept. 2018.
- [35] S. Kadekodi, F. Maturana, S. J. Subramanya, J. Yang, K. Rashmi, and G. R. Ganger. {PACEMAKER}: Avoiding heart attacks in storage clusters with disk-adaptive redundancy. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20}*, pages 369–385, 2020.
- [36] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.
- [37] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.
- [38] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Transactions on Storage (TOS)*, 13(3):1–34, 2017.
- [39] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SoCC*, 2014.
- [40] J. Li and C. Zhang. Distributed hosting of web content with erasure coding and unequal weight assignment. In *2004 IEEE International Conference on Multimedia and Expo (ICME) (IEEE Cat. No.04TH8763)*, volume 3, pages 2087–2090 Vol.3, 2004.
- [41] K. Liu, J. Peng, J. Wang, and J. Pan. Optimal caching for low latency in distributed coded storage systems. *arXiv preprint arXiv:2012.03005*, 2020.
- [42] R. S. Liu, C. L. Yang, C. H. Li, and G. Y. Chen. Duracache: A durable ssd cache using mlc nand flash. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–6, 2013.
- [43] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *SIGCOMM Comput. Commun. Rev.*, 45(3):52–66, July 2015.
- [44] V. Martina, M. Garetto, and E. Leonardi. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM*, 2014.
- [45] S. McAllister, B. Berg, J. Tutuncu-Macias, J. Yang, S. Gunasekar, J. Lu, D. S. Berger, N. Beckmann, and G. R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP ’21*, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery.

- [46] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407, 2018.
- [47] V. Mirrokni, M. Thorup, and M. Zadimoghaddam. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 587–604, 2018.
- [48] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook’s warm BLOB storage system. In *USENIX OSDI*, pages 383–398, 2014.
- [49] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at Facebook. In *USENIX NSDI*, pages 385–398, 2013.
- [50] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai Network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.
- [51] Y. Oh, J. Choi, D. Lee, and S. H. Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST*, volume 12, 2012.
- [52] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, 1988.
- [53] L. Perron and V. Furnon. OR-tools. <https://developers.google.com/optimization/>.
- [54] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A hitchhiker’s guide to fast and efficient data reconstruction in erasure-coded data centers. In *SIGCOMM*, 2015.
- [55] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *USENIX OSDI*, pages 401–417, 2016.
- [56] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, 2013.
- [57] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *FAST*, 2003.
- [58] E. Rocca. Running Wikipedia.org, June 2016. available https://www.mediawiki.org/wiki/File:WMF_Traffic_Varnishcon_2016.pdf accessed 09/12/16.
- [59] R. K. Sahoo, M. S. Squillante, A. Sivasubramaniam, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *International Conference on Dependable Systems and Networks, 2004*, pages 772–781, 2004.
- [60] L. Saino, I. Psaras, and G. Pavlou. Understanding sharded caching systems. In *IEEE INFOCOM*, pages 1–9, 2016.
- [61] Sanjay Sane. Latency and wear-out in facebook’s cdn due to ssd write pressure. Private conversation,, 7 2019.
- [62] P. Sarkar and J. H. Hartman. Hint-based cooperative caching. *ACM Transactions on Computer Systems (TOCS)*, 18(4):387–419, 2000.
- [63] M. Saxena, M. M. Swift, and Y. Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys ’12*, page 267–280, New York, NY, USA, 2012. Association for Computing Machinery.
- [64] K. Schomp, O. Bhardwaj, E. Kurdoglu, M. Muhamen, and R. K. Sitaraman. Akamai DNS: Providing authoritative answers to the world’s queries. In *ACM SIGCOMM*, pages 465–478, 2020.
- [65] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing*, 7(4):337–350, 2009.
- [66] Z. Shen, F. Chen, Y. Jia, and Z. Shao. Optimizing flash-based key-value cache systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association.
- [67] Z. Shen, F. Chen, Y. Jia, and Z. Shao. Didocache: an integration of device and application for flash-based key-value caching. *ACM Transactions on Storage (TOS)*, 14(3):1–32, 2018.
- [68] A. Sundarajan, M. Feng, M. Kasbekar, and R. K. Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *ACM CoNEXT*, pages 55–67, 2017.
- [69] A. Sundarajan, M. Kasbekar, R. K. Sitaraman, and S. Shukla. Midgress-aware traffic provisioning for content delivery. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 543–557, 2020.

- [70] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. Ripq: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015.
- [71] F. Velázquez, K. Lyngstøl, T. Fog Heen, and J. Renard. *The Varnish Book for Varnish 4.0*. Varnish Software AS, March 2016.
- [72] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, 2017.
- [73] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient mrc construction with shards. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, 2015.
- [74] L. Wang, K. Park, R. Pang, V. S. Pai, and L. L. Peterson. Reliability and security in the codeen content distribution network. In *USENIX ATC*, pages 171–184, 2004.
- [75] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS*, 2002.
- [76] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.
- [77] G. Wu and X. He. Reducing ssd read latency via nand flash program and erase suspension. In *FAST*, volume 12, pages 10–10, 2012.
- [78] K. Wu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Towards an unwritten contract of intel optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.
- [79] F. Xu, Y. Wang, and X. Ma. Online encoding for erasure-coded distributed storage systems. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 338–342, 2017.
- [80] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Trans. Storage*, 13(3), Oct. 2017.
- [81] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, Nov. 2020.
- [82] J. Yang, Y. Yue, and K. V. Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Trans. Storage*, 17(3), aug 2021.
- [83] J. Yang, Y. Yue, and R. Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, Apr. 2021.
- [84] M. M. T. Yiu, H. H. W. Chan, and P. P. C. Lee. Erasure coding for small objects in in-memory kv storage. In *Proceedings of the 10th ACM International Systems and Storage Conference, SYSTOR ’17*, New York, NY, USA, 2017. Association for Computing Machinery.
- [85] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does erasure coding have a role to play in my data center? Technical Report Microsoft Research MSR-TR-2010, 2010.

10 Supplemental information

10.1 Cluster size distribution

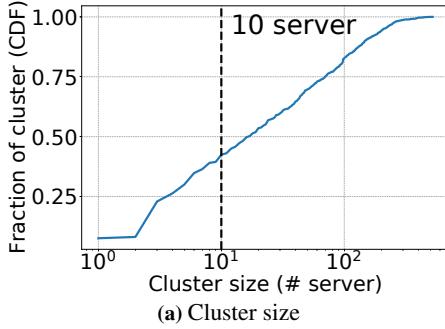


Figure 10: Cluster size ranges from 1 to over 500 servers.

10.2 Proof details

Proof of Theorem 1.

Let T_c^i denote the *characteristic time* [23, 26] of the cache at server i , given capacity C , where the characteristic time of an LRU cache measures how long it takes for a newly requested chunk to get evicted. We first prove that for any two servers i and j , T_c^i and T_c^j are nearly the same. More precisely, we show that for any $i \neq j$, $\text{Prob}(|T_c^i - T_c^j| \geq \varepsilon)$ is at most $O(W/(\varepsilon^2 C))$, using a mathematical argument similar to [60]. Where C denotes the cache size of each server and W is the variance of the write load imbalance across the servers in the cluster. Due to parity rebalancing in C2DN, $W \rightarrow 0$. So, this probability $O(W/(\varepsilon^2 C))$ vanishes as the cache size grows large.

In C2DN, when an object is requested, its chunks x_1, \dots, x_n are requested at the same time from the individual servers. Since the characteristic time of the servers that these chunks reside in are (nearly) the same as shown above, it follows that these chunks are evicted from these caches at (nearly) the same time. Thus, the chunks x_1, \dots, x_n of an object enter and exit their individual caches in a synchronized way, even though there is no explicit coordination among the caches.

10.3 Additional details on parity rebalance

Here, we give more details about the bucket assignment algorithm discussed in §4 under the three scenarios (1) Initial bucket assignment, (2) Server failure, (3) Server addition. We first consider the case where the servers are homogeneous and later extend the algorithm to the heterogeneous case.

Initial bucket assignment. The algorithm runs in two phases. In the first phase, the data buckets are assigned to the servers using the consistent hashing algorithm. The algorithm chooses K consecutive servers on the consistent hash ring

from the bucket's hash location in a clockwise direction to assign the K data chunk buckets. In the second phase, the parity chunks are assigned to the servers such that the load is evenly balanced across the servers. The second phase is described in Algorithm 1.

Algorithm 1 Phase 2. Parity rebalance

```

1: Input : Set of available servers  $\mathcal{A}$  and the total write load on the
   cluster  $W$ .
2: Set of  $\mathcal{N}$  parity buckets. For  $n \in \mathcal{N}$ , the sum of sizes of the
   parity chunks in the bucket is  $s_n$ .
3: A set  $L$  with  $l_i$  denoting the current write load (due to assignment
   of data buckets and uncoded objects) on server  $i$ .
4: Output : A valid assignment of parity bucket to the servers.
5: Initialize : A set of vertices  $V \leftarrow \emptyset$ , a set of edges  $E \leftarrow \emptyset$ , an
   empty graph  $\mathcal{G} = (V, E)$ .
6: Add source node  $S$ , terminal node  $T$ , nodes corresponding to
   parity buckets and available servers to  $V$ .
7: for  $n \in \mathcal{N}$  do // Loop through parity buckets
8:    $e \leftarrow ((S, n), n_s)$  // Create edge between nodes between
   source-node  $S$  and bucket-node  $n$  with weight equal to size
   of the bucket  $n_s$ 
9:    $V \leftarrow n, E \leftarrow e$ 
10:  for  $a \in \mathcal{A}$  do
11:    if data chunk of bucket  $n$  is not assigned to  $a$  then
12:       $e \leftarrow ((n, a), n_s)$  // Create edge between parity bucket
   node  $n$  and available server  $a$  with size of parity bucket  $n_s$ .
13:    end if
14:  end for
15: end for

16: for  $a \in \mathcal{A}$  do // Loop through available servers
17:    $c_a \leftarrow \max(\lceil \frac{W}{|\mathcal{A}|} \rceil - l_i, 0)$  // Available budget on each server
18:    $e \leftarrow ((a, T), c_a)$  // Create edge between server nodes  $a$  and
   sink-node  $T$  with weight  $c_a$ 
19:    $V \leftarrow v, E \leftarrow e$ 
20: end for

21: Run MaxFlow( $S, T$ ) between the source-node  $S$  and terminal
   node  $T$ .
22: for Each parity bucket  $n \in \mathcal{N}$  do
23:   Assign the parity bucket to the least loaded server that has a
   positive assigned flow from the bucket.
24: end for

```

Post running the Phase 1 of the algorithm, the available write budget on each server a (c_a) in line 17 of Algorithm 1, is obtained as follows. If the total unique bytes requested to the cluster is W , then each server should host traffic not more than $\lceil \frac{W}{|\mathcal{A}|} \rceil$ bytes. After phase 1 of the algorithm, if l_i is the traffic assigned to server i , then the available budget on server i i.e., c_i is obtained as $\max(\lceil \frac{W}{|\mathcal{A}|} \rceil - l_i, 0)$. An empty graph is initialized in line 5 and the source and terminal nodes are added in line 6. In line 8, for each bucket n we add an edge from the source node S to the bucket-node with a capacity

that equals the size of the parity bucket. Through lines 10-14, for each bucket n add a directed edge from the node that corresponds to the parity bucket to a server-node that is not assigned a data chunk of bucket n . These edges capture if the parity bucket can be assigned to a server. Then we assign these edges with a capacity that equals to the size of the parity bucket. In lines 16-19 directed edges are added from server nodes (a) to the sink node with a capacity c_a i.e., the available write budget on the server. Now run the max-flow algorithm in the graph between the source-node S and the sink-node T to find a valid assignment of parity buckets to the servers. To assign a parity bucket to a server, we find edges from the parity-node to the server-nodes that are assigned a positive flow. The servers are potential candidates for assignment. Empirically, we find that in most runs, the algorithm finds a single candidate server, if not, we assign the parity bucket to the least loaded server among the potential candidates.

Server failure. When a server fails the data buckets (D) and parity buckets (P) belonging to the server need to be reassigned. As done previously, the data buckets are reassigned using the consistent hash ring. Now, the new data bucket allocation could invalidate some of the previous parity bucket assignments (on the currently available servers) as the data chunks and parity chunks cannot cohabit the same server. Let the invalidated parity buckets be P' . The available budget c_i of each server i is recalculated using the total traffic W and the number of available servers $|\mathcal{A}|$. Now, the buckets $P \cup P'$ (parity buckets of failed server and invalidated parity buckets) are assigned to the servers using Algorithm 1 by reassigning corresponding capacities in line 9. When a server fails we also keep track of the parity buckets that were assigned to it before failure. Some of these buckets could be reassigned to the server when it is available again depending on the available write budget at each server.

Server addition. When a server is available again, it gets assigned the data buckets using the consistent hashing algorithm. We recompute the capacity of each server as done previously. Now, as we have kept track of the parity buckets the server was assigned prior to failure, we try to re-assign as many of those parity buckets as possible. If we get back all buckets and we still have capacity for more buckets to be assigned, we iteratively pick parity buckets from the most loaded server.

Algorithm complexity. In Algorithm 1 the cost of constructing the graph can be computed using $O(|T| \times |\mathcal{N}|)$ time complexity where T is the number of servers. And the time complexity is because we need to decide if we wish to add an edge between the parity chunk and the server. Further, the runtime complexity of the MaxFlow algorithm in line 21 is computed as follows. The total available budget on the servers is $B = \sum_{i \in S} \left(\left\lceil \frac{W}{|\mathcal{A}|} \right\rceil - l_i \right)$. Then, the runtime complexity is $B \times |\mathcal{N}| \times |T|$.

Heterogeneous servers. We extend consistent-hashing-based bucket assignment to the case of heterogeneous servers. Each

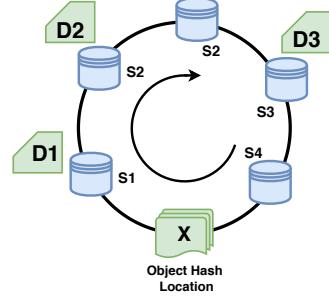


Figure 11: Consistent Hashing for a cluster with heterogeneous servers. The data buckets D_1 , D_2 and D_3 are hashed to unique servers S_1 , S_2 and S_3 .

server is represented by at least one virtual node on the consistent hashing ring. The number of virtual nodes added is proportional to the capacity of the server (e.g., if server A is $2 \times$ larger than server B, then server A would have $2 \times$ as many virtual nodes).

The data buckets are mapped to the consistent hashing ring as follows. From the bucket's hashed position on the ring, we move along the ring in a clockwise direction and assign — the K data buckets — iteratively to the virtual nodes encountered. However, while assigning, we step over servers that have been assigned any of the other K data buckets. This ensures each of the K data buckets are assigned to different servers. Figure 11 shows a placement example. The cluster consists of 4 servers S_1 , S_2 , S_3 and S_4 of capacity C , $2C$, C , C respectively. Virtual nodes are indicated by the server name. Note that, as S_2 is twice the capacity of the servers, we create two virtual nodes for S_2 . By chance, we assume that the two virtual nodes hash to adjacent positions on the ring. Now, if the bucket is hashed to a location X on the consistent hash ring, then the data buckets D_1 , D_2 and D_3 are assigned to the first three servers encountered by moving in the clockwise direction from X . This is S_1 , S_2 (S_2 again and thus skipped), and S_3 , which is the resulting bucket placement. After data buckets are assigned, we use Algorithm 1 to assign parity buckets.

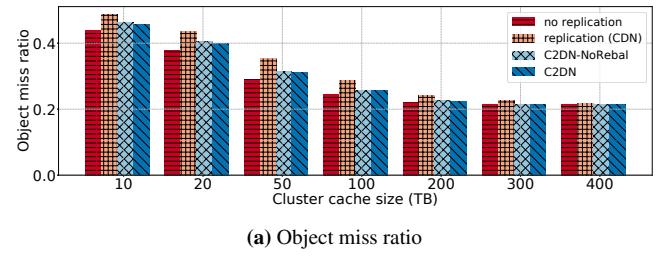


Figure 12: Object miss ratio of different systems.

10.4 Additional figures for sensitivity analysis

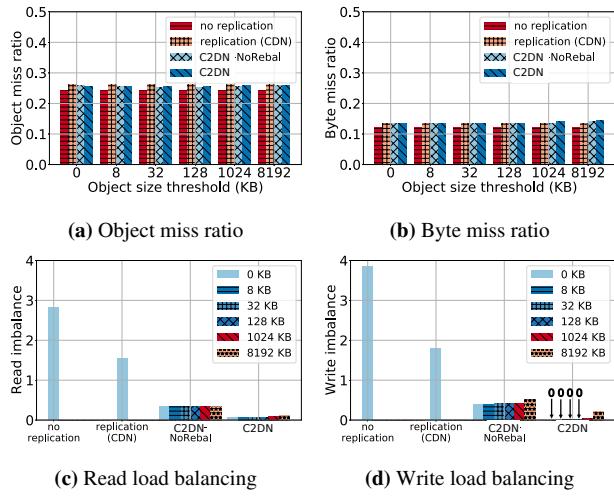


Figure 13: Impact of coding size threshold on miss ratio and load balancing.

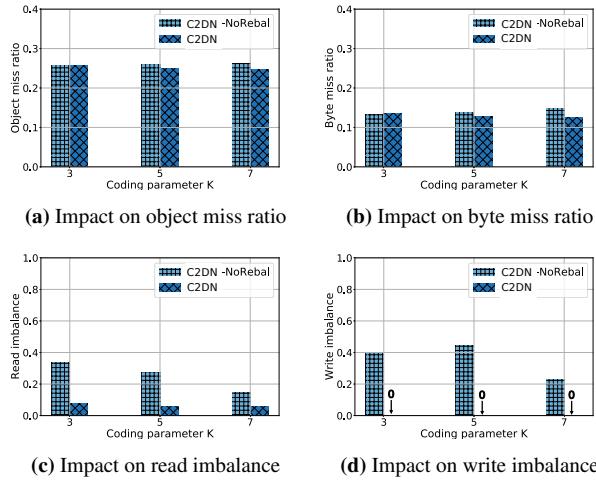


Figure 14: Impact of parameter K on miss ratio and load imbalance.

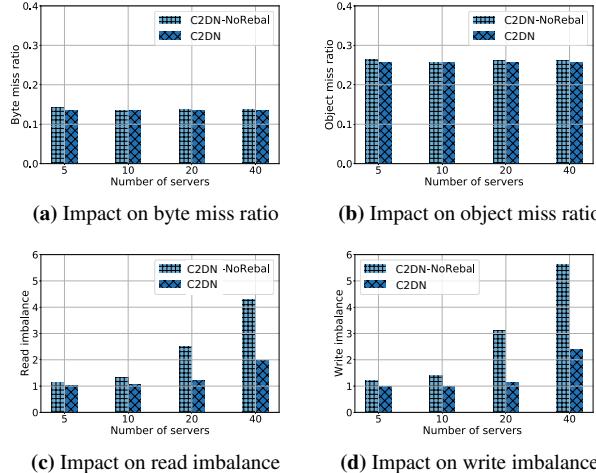


Figure 15: Impact of number of servers on miss ratio and load imbalance.

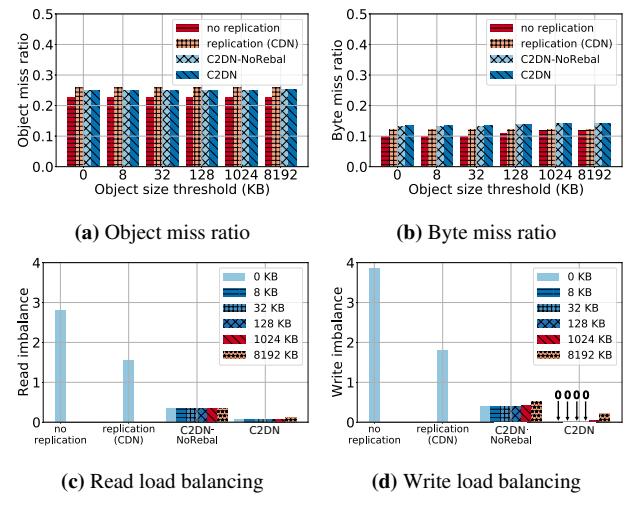


Figure 16: Impact of coding size threshold on miss ratio and load balancing (LRU).

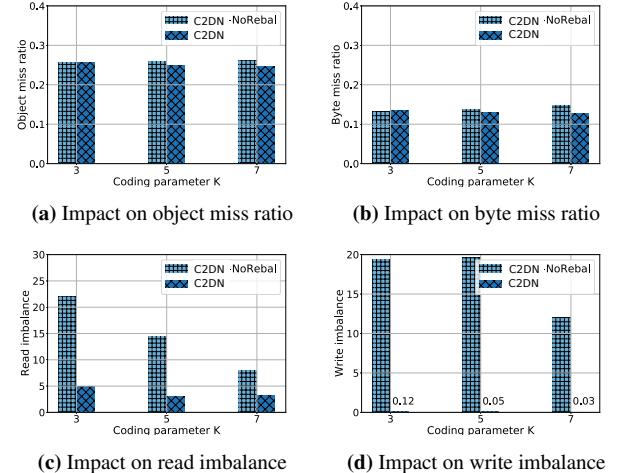


Figure 17: Impact of parameter K on miss ratio and load imbalance (LRU).

Optimizing Network Provisioning through Cooperation

Harsha Sharma*, Parth Thakkar*, Sagar Bharadwaj*, Ranjita Bhagwan, Venkata N. Padmanabhan,
Yogesh Bansal, Vijay Kumar, Kathleen Voelbel
Microsoft

Abstract

The rise of cloud-scale services has fueled a huge growth in inter-data center (DC) Wide-Area Network (WAN) traffic. As a result, cloud providers provision large amounts of WAN bandwidth at very high costs. However, the inter-DC traffic is often dominated by *first-party applications*, i.e., applications that are owned and operated by the same entity as the cloud provider. This creates a unique opportunity for the applications and the network to cooperate to optimize network provisioning (which we term as optimizing the “*provisioning plane*”), since the demands placed by dominant first-party applications often *define* the network. Such optimization is distinct from and goes beyond past work focused on the control and data planes (e.g., traffic engineering), in that it helps optimize the provisioning of network capacity and consequently helps reduce cost.

In this paper, we show how cooperation between application and network can optimize network capacity based on knowledge of the application’s deadline coupled with network link failure statistics. Using data from a tier-1 cloud provider and a large enterprise collaboration service, we show that our techniques can potentially help provision significantly lower network capacity, with savings ranging from 30% to 45%.

1 Introduction

The growth of cloud-scale mega services ¹ has fueled a huge increase in network traffic, not only within data centers (DCs) and on the Internet but, importantly, also on the inter-DC wide-area network (WAN). We observe that the inter-DC WAN traffic for large public cloud providers is dominated by first-party applications and services e.g., the e-commerce service in the case of Amazon, Google search and Gmail in the case of Google, and Bing search and Office 365 in the case of Microsoft [8, 9]. This presents both a challenge and an opportunity.

The challenge is that, as application usage grows, so does the inter-DC WAN traffic. This leads to increased provisioning of inter-DC WAN capacity, often with a multiplicative factor to provide redundancy. Note that large cloud providers typically build and operate their own inter-DC WAN with owned and leased fiber, which is different from a small provider who might use an ISP for such connectivity. The increased availability of bandwidth, and the fact that it is sunk cost (i.e., already paid for and so might as well be utilized), fuels the appetite of existing and new applications. They consume more bandwidth (though there might be back-pressure due to mechanisms such as traffic engineering [7, 8] or internal pricing), causing the network to forecast higher usage for the future, which in turn leads to increased network provisioning and so forth. Such a vicious cycle is quite expensive since inter-DC WAN bandwidth imposes an amortized annual cost of 100s of millions of dollars on a large cloud provider [7].

This situation is quite different from the third-party application setting, wherein market forces of cost and price operating between the consumer of bandwidth (third-party application) and the provider of bandwidth (cloud provider) would tend to keep the process in check. If the provider has excess capacity, they could find new customers to sell it to rather than encouraging existing customers to be profligate in their use of bandwidth. There is an opportunity to do better in the first-party case by leveraging *cooperation* across the network and the applications. Specifically, rich information flow between the application and the network would enable informed network provisioning. We term the task of provisioning network capacity as optimizing the “*provisioning plane*”, to set it apart from the well-established notions of the control and data planes.

We leverage the knowledge of the *application’s deadline* to optimize network provisioning. Such an application deadline is quite distinct from the more common notion of delay tolerance, which centers on the latency that the application can tolerate in network communication. Application deadline, on the other hand, arises from the application’s ability to pause, or defer, some or all of its activities and the associated com-

*Equal contribution

¹We use “services” and “applications” interchangeably.

munication for an extended period of time, for example, when there is a loss of network capacity because of link failures. If the network capacity is likely to be restored (e.g., by repairing the failed link(s)) within the period of the application deadline, then we can dispense with the provisioning of redundant network capacity for “deferrable” traffic.

If user impact is to be avoided, the deferrable application activity, and the resulting traffic, should be in the background. A good example is load balancing, which might be triggered when a server is at the risk of running “hot” on one or more resources such as CPU, IO (I/O operations) capacity, or storage space, and therefore needs to shed load. Load balancing would typically involve the transfer of a large volume of data (e.g., user mailboxes in the context of an email application). This activity could be paused for a length of time so long as there is enough “headroom” in resources, i.e., none is on the verge of being saturated. Based on past data, we can make a conservative but specific choice of deadline to facilitate network provisioning.

We develop a generic framework for applications to express their demands in a way that reflects the traffic volume, deadlines, and desired probability of satisfaction (i.e., the likelihood that the demand will be met). We also develop a model for link failures and repairs that is informed by historical data. A key aspect of this model is the data-driven discovery of links with correlated failures, say because they share one or more components (e.g., fiber conduit, power supply, etc.) and the recreation of such correlated failures to simulate scenarios that are particularly challenging from the viewpoint of capacity provisioning.

We evaluate the above using data from the WAN of a tier-1 cloud provider, Microsoft, and from a large enterprise collaboration service, M365 Substrate (which we shall hereafter refer to as Substrate), which uses this WAN. We find that application-informed provisioning reduces network capacity by more than 30% in multiple regions.

Note: While we use data from commercial services — Microsoft and Substrate — to drive our analyses here, this data is highly sensitive due to commercial reasons. Therefore, we are not in a position to report metrics such as the traffic volume, network capacity, failure characteristics, etc. in an absolute sense and report relative numbers instead.

1.1 Comparison with Traffic Engineering

Before proceeding, it is useful to compare our work on cooperative provisioning with the well-established prior work on traffic engineering.

Traffic engineering focuses on supporting a specified demand (i.e., source-destination flows, quality of service requirements) on a given network (i.e., the network topology, including link capacities). Both the demand matrix and the network topology are taken as input and traffic engineering looks for ways of supporting the demand at *run time* through

techniques such as routing and path selection [8], smoothing to shift traffic peaks into the valleys [7], and using store-and-forward techniques to deal with temporal offsets between traffic peaks in different parts of the network [13]. There is a body of traffic engineering work that specifically considers the problem of satisfying deadlines in the face of new demands, link failures, etc., by employing admission control, online scheduling, and fairness across demands when the deadlines cannot be satisfied [11, 22].

In comparison, our work on cooperative provisioning focuses on the *planning phase* that precedes the creation of the network or the augmentation of its capacity. This requires modeling link failures upfront by simulating the failures of (combinations of) links and making sure that sufficient capacity is provisioned to accommodate the demands even in the face of such failures. In contrast, traffic engineering deals with link failures as these arise and does not entail provisioning capacity. Furthermore, in optimizing the provisioning of network capacity, the ability to defer traffic by pausing workload, for days or weeks, provides us a qualitatively greater flexibility for optimization than the QoS requirements that are typically dealt with in traffic engineering [11, 22]. For instance, the former would allow tiding over link failures without redundant provisioning, while the latter typically would not.

The opportunity to optimize provisioning by counting on the ability to pause certain demands arises because of the first-party setting, which enables cooperation between the network and the applications. Therefore, cooperative provisioning is limited to settings where such cooperation is feasible. Traffic engineering, on the other hand, is applied more broadly (e.g., in ISP networks) and as such cannot assume such cooperation.

2 Application Traffic Demands

The starting point for network capacity provisioning is the demand placed by applications. Since building a network or augmenting the capacity of an existing network would typically take time (several months to more than a year), there is also the need to *forecast* future demand. Forecasting is a well-studied problem and there exist many techniques for it [15, 19]. Thus, in this paper, we do not focus on forecasting and our analysis is based on taking a snapshot of the present demand and using it to perform capacity planning, with and without our optimizations. However we do show in Section 4.3 that our findings carry over even if applied to future demand obtained through forecasting.

2.1 Demand Specification

An application, i , specifies its demands as a set of discrete time series, with the j^{th} demand on behalf of i being specified as $D_{ij} = (t, A, B, V, d, p)$, i.e., a demand for conveying a volume V of traffic from location A to location B , expressed at time

t and with a deadline of d (that is, the demand should be satisfied during $[t, t+d]$), and with the desire that the demand be satisfied with a probability of p , i.e., in the network failure scenarios that cumulatively account for a fraction p of time. (Each of these quantities is set by the application i and should carry the subscript ij , but we drop it for the ease of exposition.) We describe two types of application demands which are relevant to this paper: immediate and deferrable.

Immediate: Such traffic tends to be in the critical path of user latency, so there is little temporal flexibility. The deadline for such a demand is “now”, so the demand can be expressed as a data rate to be supported between a specified source and destination. Hence we can simplify the formulation of the demand as $D_i = (t, A, B, r, p)$, where we subsume V and d by the rate $r = V/d$. Note that if the network were provisioned to support the desired rate r for this demand continually, we would be able to transfer volume V within a deadline d .

Deferrable: The traffic demand arising from asynchronous activity tends to be temporally flexible, i.e., deferrable, with a long deadline. A good example is traffic arising from distributed load balancing, wherein resources on a server (e.g., CPU, IO capacity, storage space) grow in terms of utilization (i.e., start becoming “hot”), necessitating the rebalancing of the workload and entailing the transfer of related data (e.g., a user’s mailbox in the context of an email service or folder in the case of a storage service). The urgency of the load balancing activity and hence the deadline of the resulting traffic demand would depend on how much headroom there is on the server resources that are heating up.

Deferrable demand would be specified with a deadline d that reflects the degree of temporal flexibility. For instance, in the case of load balancing triggered workload, there might be enough headroom in the server resources (i.e., none is close to being saturated) to allow d to be set to days or even weeks. Note that the demand can tolerate such latency (e.g., by slowing, pausing or turning off the application components responsible for the demand); however, once the application component is resumed and it actually starts its data transfers, these would be completed in a much shorter and typical “network timescale” (e.g., within seconds or minutes or hours, depending on the size of the transfer). Also, just because a demand can be temporarily paused, it does not mean that it can be forgotten. So, the network would be provisioned with sufficient capacity to allow “catch-up” on the deferred demand once it is resumed. Processes that perform periodic tasks on data such as garbage collection, compliance checks, and workload analytics can also cause asynchronous traffic demands.

3 Cooperative Provisioning

In this section, we describe how the network, with the application’s cooperation, can provision the network efficiently. We

first provide some background on network provisioning and how it is currently done. Next, we discuss two specific opportunities for cooperative provisioning: (a) deriving a smoothed demand signal based on explicit application input, and (b) provisioning network redundancy in a way that is cognizant of the demand deadlines and the repair time of links. Finally, we describe a mathematical framework which, using constraint optimization, ensures appropriate provisioning of network capacity, including redundant capacity, to satisfy all demands. We dub this framework *Approv*, short for “Application-informed Provisioning”.

3.1 Background on Capacity Provisioning

We sketch how capacity provisioning is done by the cloud provider, Microsoft. Although we do not have information from other providers, we believe that the process outlined here is general and not provider-specific.

Periodically, e.g., every few months, the cloud operator forecasts demands between each datacenter (DC) pair and subsequently provisions network capacity to satisfy all demand forecasts. Since the demands arise from third-party applications (which the operator has no real leverage over) as well as first-party ones, the network typically works with just the actual traffic originated by applications (i.e., an implicit signal) rather than an explicit expression of application traffic demand (e.g., as in Section 2.1). Such an implicit signal is derived from application traffic categorized into tiers of service, with each tier corresponding to a different priority level [7, 8].

Based on this implicit demand signal derived from actual traffic, the operator derives the peak or 95%ile (P95) traffic level and uses this to forecast the traffic level, say months or even more than a year into the future. In the absence of any additional context from the applications sourcing the traffic, the operator has no choice but to work with the implicit demand signal, no matter how spiky it is (i.e., how high the peaks are).

With the demand so determined, the operator proceeds to simulate the failure of various links and combinations of links. A simulator is used to route each demand over possibly multiple paths chosen based on the latency and the available bandwidth. If the demands cannot be satisfied using the available bandwidth, the operator would augment capacity in the network (i.e., add redundant capacity) so that the demands are satisfied even in the face of such failures. Specifically, the capacity of a link is augmented if its utilization during simulation rises above a high-water mark and it is reduced if the utilization drops below a low-water mark. At the end of this iterative process of simulation, the operator would arrive at the *network capacity build out plan*, specifying the number and capacities of the links needed.

There are a couple of details worth noting. First, in general, the operator would start with an existing network and then determine the capacity augmentation and link decommission-

ing needed based on the demand forecast. However, to enable fair comparisons in this paper, we provision the network from scratch, i.e., starting with links of zero capacity, which helps avoid the need to carry the baggage of past provisioning done without the benefit of Approv. Second, for simplicity, we work with the specified topology (where each link starts out with zero capacity) and only compute the capacity needed on the links that are part of the topology. While the Approv framework is general enough to accommodate the creation of new links, the ability to create a link between two locations would, in general, be constrained by practical considerations that would have to be provided as additional input.

3.2 Leveraging Application Cooperation

We can leverage the application’s cooperation — specifically that of first-party applications that have nothing to hide from the network — to improve the network capacity provisioning process outlined above in two ways.

3.2.1 Explicit Demand Signal to Aid Smoothing

First, rather than just work with the implicit signal of actual traffic sent, the operator can take advantage of application demands expressed explicitly per the framework presented in Section 2.1. Such explicit knowledge would enable the operator to calculate the smoothed demand.

For example, even if the peak (or P95) rate of traffic sent by an application was 1Gbps, knowledge of the explicit application demand (including the deadline d from Section 2.1) might allow the operator to determine that the application’s traffic could have been smoothed down to a peak rate of 0.8 Gbps. Therefore, instead of basing its forecast, and consequently the actual capacity provisioning computation, on the 1Gbps peak, the operator could work with the smoothed 0.8 Gbps peak, thereby “rightsizing” the capacity. In the case of a deferrable demand, expressing the demand in terms of the volume V and deadline d would enable further rightsizing of capacity compared to just smoothing down the demand expressed as a rate r . The reason is that much or all of a demand in terms of V and d could be fit within the valleys and headroom of the immediate demands (Figure 2), obviating the need for any additional provisioning.

3.2.2 Rightsizing Redundancy for Deferrable Demands

Second, knowledge of the deadlines enables the operator to “rightsizing” the redundancy too. The intuition is that if an application demand has a longer deadline than the time to recover a failed link, then during capacity provisioning (and during the subsequent operation of the provisioned network), we can assume that part or all of such demands are simply paused or deferred until the failed link has been repaired. This would avoid the need for provisioning full redundancy for such deferrable demands.

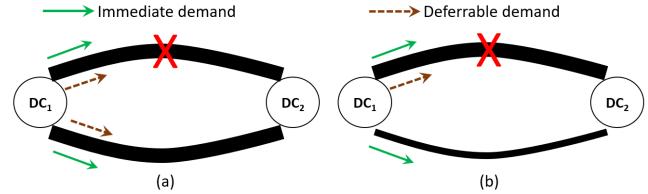


Figure 1: Rightsizing of redundancy provisioning: (a) shows the default provisioning of redundancy with no distinction between immediate and deferrable demands. Upon failure of the primary link at the top, the backup link at the bottom has the capacity to carry the full load of traffic. (b) shows how the provisioning of redundancy reduces by confining it to the immediate demand. The thinner backup link is sized to carry just immediate demand, although the overall provisioning would be sized to accommodate the “catch-up” of the deferred demand upon repair.

A simple example illustrates the savings made possible by such informed redundancy provisioning. Consider a network comprising just two nodes, A and B . Consider two cases:

Case I: The application’s implicit demand is 1 GB/day.

Case II: The application explicitly specifies a demand of $V = 30$ GB with a deadline of $d = 30$ days, which also works out to 1 GB/day.

Say the cloud operator knows that the $A - B$ link could be down for up to 10% of the time, i.e., for up to 3 days in the month. Therefore, to support the implicit demand in Case I, the operator would have to provision at least one additional link of 1 GB/day capacity (and possibly more depending on the likelihood of concurrent failures of multiple links). Therefore, the operator would have to provision a total of at least $1 + 1 = 2$ GB/day capacity, and possibly more, on the $A - B$ path.

On the other hand, in Case II, explicit knowledge of the deadline would enable the operator to determine that the demand could be paused, or deferred, temporarily to “tide over” the network link’s downtime. Of course, once the link has been restored, there would be the need to “catch up” on the deferred demand. Still, given the up to 10% downtime of the link, the operator can get away with provisioning a single $A - B$ link of capacity 1.11 GB/day (computed as $1/(1-0.1)$), which would be significantly lower than the (at least) 2 GB/day in Case I above. (In this illustrative example, we ignore the quantization of bandwidth, i.e., the minimum step size for allocation.)

3.2.3 Summary

In general, we would have a mix of “immediate” user-facing demand and “deferrable” background demand, and the former cannot tolerate any delay. Therefore, we might still have to include redundant links. However, by rightsizing the provisioning of redundancy for the deferrable demand, we would

be able to reduce the overall capacity provisioned, as illustrated in the simple example in Figure 1. Furthermore, as explained above, specifying deferrable demand in terms of volume and deadline would enable more effective provisioning (perhaps even fitting within the valleys and headroom of the immediate demand) than working with a smoothed rate.

In the remainder of this section, we formalize our framework for cooperative capacity provisioning and also present our method for simulating network link failures and repairs (including the concurrent failure of links that carry shared risks), informed by the history of actual link failures and repairs.

3.3 Framework for Capacity Provisioning

Our Approv framework models network provisioning as a constraint optimization problem using a linear program (LP). As with any network provisioning approach (Section 3.1 provided some background on this), the LP needs to simulate all “likely” link failures, singly or in combination, and ensure that the network has sufficient redundant capacity to fulfill all demands despite such link failures.

A key challenge in our work arises from our richer demand model compared to prior work. State-of-the-art approaches [1] take demand data rates as input (e.g., 1 Gbps from A to B) and then only simulate link failures as point-in-time events, to verify the satisfaction of demand in the face of failures. Since we support demands that specify deadlines (d) that could stretch to days or even weeks, we incorporate a much richer notion of failures, including the actual duration, i.e., the time from the loss of capacity due to the onset of a failure episode until the restoration of capacity upon repair. As we discuss, this approach enables optimizations that are not possible with point-in-time simulation of failures.

Given a time-window (t_s, t_e) , We define a *failure scenario* as a set of per-link time-series $\{f_1, f_2, \dots, f_n\}$, where n is the number of links in the network. f_l is a time series representing the status of link l , i.e., whether it was up (working) or down (failed), over time. f_l is a time series, $f_{l1}, f_{l2}, \dots, f_{lt}$ over the full duration of the simulation, where $f_{lt} \in \{0(\text{down}), 1(\text{up})\}$ and time is discretized (in steps of 1 hour in our work). To synthesize realistic failure scenarios that extend over the full duration of the simulation, we have built a novel history-based failure model, described in Section 3.4.

The Approv capacity provisioning formulation assumes a fixed network topology, where the nodes are either datacenters in the cloud network or network points of presence. Given such a topology and the application demands the Approv provisioning framework allocates capacity to each link in the topology so as to satisfy all demands, by simulating two functions:

- *Topology and Routing:* Approv incorporates constraints arising from the network topology and the set of valid network-level routes between any two datacenters (DCs). The routes

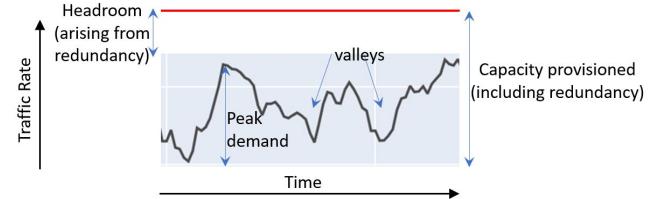


Figure 2: Illustration of the peaks and valleys in the time series of the immediate demand, and of the headroom above the peak, arising from the overall capacity provisioned, inclusive of the redundancy. The valleys and the headroom represent capacity that could be used to accommodate background demand, subject to its deadline.

are derived from the large inter-DC WAN deployed by Microsoft.

- *Link Failures:* Approv also incorporates constraints to capture the effect of each failure scenario that is simulated. This helps ensure that the network is provisioned with enough capacity to fulfill all demands even in the face of such failures.

For ease of exposition, we use a uniform p (i.e., probability of satisfaction) for all demands. Later, in Section 3.3.2, we relax this assumption to generalize the framework to incorporate a demand-specific p .

Given a set of failure scenarios \mathbf{F} , we construct a “capacity provisioning LP” with constraints corresponding to each combination of demand and failure scenario. The LP generates a capacity provisioning plan in two steps:

1. *Provisioning of immediate demands:* Such demands have an immediate deadline, so we represent these in terms of the rate r , as discussed in Section 2.1. The demanded rate needs to be supported even in the face of the failure scenario considered. To ensure this, we sweep across time, from the start to the end of the simulation. We ensure that the network is sufficiently provisioned to support the immediate demands at each point in time, which corresponds to a specific combination of link failures.
2. *Provisioning of deferrable demands:* Deferrable demands are expressed in terms of the desired volume, V , of data to be transferred and the corresponding deadline d . Our provisioning framework first checks to see whether (and if so, how much of) such demands could be accommodated in the valleys and the headroom of the capacity provisioned above for the immediate demands (see Figure 2). To the extent the deferrable demand exceeds what can be so accommodated, the framework augments the capacity on one or more links to ensure that the deferrable demands are satisfied too (in addition to the immediate demands). In doing so, we take advantage of the demand smoothing and redundancy rightsizing techniques discussed in Section 3.2.

At the end, we arrive at the network build plan, which gives the capacity to be provisioned on each link in the network. We present the LP formulation and an example in Appendix A.1 and A.2.

3.3.1 Accommodating Probability of Satisfaction (p)

To ensure that the capacity provisioned in the network build plan satisfies the demand with the desired probability p , we use the failure model (discussed next in Section 3.4) to generate a large number of failure scenarios. As discussed in Section 3.4, the generation of failure scenarios is governed by the failure and repair history of each link and group of links. Consequently, the more likely link failure combinations will appear more often in the failure scenarios, just as we would desire. Since we would like the network to be provisioned so as to satisfy the demand with probability p , or equivalently in at least a fraction p of the failure scenarios considered, we sort the $|F|$ individual failure scenarios in increasing order of the capacity of the build plan generated when each such scenario alone is simulated. To enable the sorting, we consider the impact of each failure scenario on the build plan capacity in isolation instead of the collective impact of a set of failure scenarios, as in the capacity provisioning LP discussed in Section 3.3. We then consider just the first p fraction of the failure scenarios (i.e., the $p|F|$ scenarios with the least impact on capacity) and disregard (or cut off) the last $1 - p$ fraction of scenarios (i.e., the ones with the greatest impact on capacity). This subset of failure scenarios is then provided as input to the capacity provisioning LP to generate the build plan.

3.3.2 Accommodating Demand-Specific p

In general, each demand D_i could have its own p_i , representing the desired probability of satisfaction for that demand. To accommodate such demand-specific levels of p , we employ an iterative process. We first sort the p_i in increasing order, i.e., going from the least level of assurance sought by a demand to the highest. For ease of exposition, we assume that the sorted order is p_1, p_2, \dots, p_n . Then, we proceed as follows, where in each subsequent step, the additional capacity provisioned, if any, is over and above that which was already provisioned in the preceding steps. In other words, the capacity provisioned never decreases as we progress through the steps:

1. First, present all demands (D_1, D_2, \dots, D_n) to Approv and sort the failure scenarios in increasing order of their *individual* impact on capacity (as outlined at the end of Section 3.3 above). Then, pick p_1 as the cutoff in this sorted list of failure scenarios (i.e., focus on just the first p_1 fraction of the failure scenarios) and run Approv on these, to arrive at the *cumulative* capacity. This would ensure that the network is provisioned to satisfy all demands with probability (at least) p_1 .

2. Then, exclude the first demand, D_1 , and present the rest (D_2, \dots, D_n) to Approv and sort the *remaining* failure scenarios (i.e., excluding the ones already satisfied in step 1 above) in increasing order of capacity impact. Then, pick p_2 as the cutoff in this sorted list of failure scenarios, and proceed as in step 1 above. This would ensure that demands (D_2, \dots, D_n) are satisfied with probability (at least) p_2 . Note that the provisioning performed in step 1 has already ensured that all demands (D_1, D_2, \dots, D_n) are satisfied *concurrently* in the face of the p_1 fraction of failure scenarios. It is only for the additional $p_2 - p_1$ fraction of failure scenarios (during which D_1 need not be satisfied) that some of the capacity provisioned to satisfy D_1 in step 1 could potentially be used to satisfy (D_2, \dots, D_n) . Therefore, this subsequent step (step 2) does not *impinge* on the provisioning done in the earlier step (step 1).
3. Proceed accordingly, progressively raising the bar on the probability of satisfaction while narrowing down the corresponding set of demands considered at each step.
4. The process would conclude when, in the last step, we only consider demand D_n and ensure its satisfaction with probability p_n .

This ensures that the provisioning at the end of the process would satisfy all demands D_i with the corresponding probability p_i .

3.4 Failure Modeling

In this section, we describe our *history-based* failure model that we use to generate realistic failure scenarios to drive the provisioning framework described in Section 3.3. Using a history of link failure characteristics, we build a generative model of link failures that captures characteristics such as the distribution of Time To Recovery (TTR) and the Time Between Failures (TBF) for the individual links, and the correlation of failure and recovery across links.

Dataset: The dataset we use to build the failure model contains detailed link failure data at hourly granularity collected over a period of 13 months for the over 500 links in Microsoft inter-DC WAN. The WAN consists of 17 interconnected regions, such as Asia-Pacific (APAC) and North America (NAM), with each region including multiple datacenters. Each link l in the network is characterized by a discrete *link-status vector* time-series f_l , where f_{lt} is 1 if the link l was up at time t and 0 if the link was in a failed state at time t .

We build a separate model for each region to keep the modeling tractable, since there could be correlations — accidental or otherwise — across arbitrary pairs of links. Accordingly, our failure modeling uses two main steps, *Link Clustering and Characterization* and *Failure Scenario Generation*. We now describe each step.

Cluster no.	Correlation	% links	Cluster no.	Correlation	% links
0	0.81	46.3	1	0.75	7.3
2	0.79	7.3	3	0.79	7.3
4	0.80	4.9	5	0.92	7.3
6	1.0	2.4	7	1.0	2.4
8	1.0	2.4	9	1.0	2.4
10	1.0	4.9	11	0.97	4.9

Table 1: Avg intracluster Pearson correlation coefficient

3.4.1 Link Clustering and Characterization

Link failures are often correlated since links could share components such as a power source, a router, or a fiber cable [16, 20]. To capture such correlations, we cluster links that display similar failure patterns in our data. We run the Complete-linkage agglomerative clustering algorithm [3] using the *1-Pearson correlation coefficient* between the link-status vectors noted above as the distance metric between two links to form such clusters of links. Links which fail at the same time will have correlated failure patterns and hence land in the same cluster (In Section 3.4.2, we explain how we simulate failure of all links in a cluster simultaneously). We evaluated the average of Pearson correlation coefficient between links within each cluster, while sweeping over the count of clusters, and determined that setting the number of clusters to 12 yielded a satisfactory division of links, with the average intra-cluster Pearson coefficient being 0.9.

Table 1 shows average intra-cluster Pearson correlation coefficient for the Asia-Pacific (APAC) region. The high intra-cluster Pearson correlation coefficient underscores the high degree of correlation in the failure pattern of links within the clusters. Note that cluster 0, containing over 46% of the links, comprises links whose failure pattern does not correlate with that of any other link; in fact, these links suffer few failures and so their link status vectors are set to 1 at almost all times. So, this is not considered as a failure cluster during the generation phase (Section 3.4.2).

Since the clusters capture links with similar link-status vectors, we assume that links belonging to a cluster have the same distribution of Time To Repair (TTR) and Time Between Failures (TBF). Accordingly, we estimate a single distribution of TTR and TBF for the links in each cluster. From the link-status vectors in a given cluster, we gather all the link TTRs and estimate the Cumulative Distribution Function (CDF) of TTR by using linear interpolation. We do likewise to estimate the CDF of the TBF. We then use these CDFs to generate plausible failure scenario, i.e., a set of realistic link-status vectors for each link.

3.4.2 Failure Scenario Generation

Our failure scenario generation algorithm uses the estimated CDFs to generate a common link-status vector for each link cluster for a given time-window. This approach assumes that links within a cluster are perfectly correlated: all links fail and are restored at exactly the same time. While simplifying

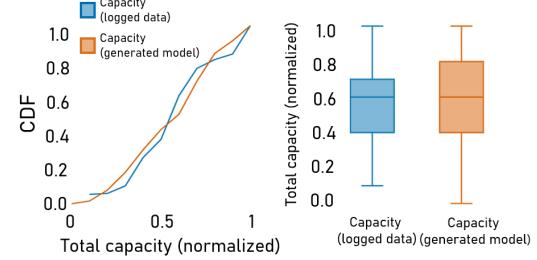


Figure 3: Comparing distribution of total capacity between logged historic data and generated failure scenarios.

the task of generating failure scenarios, this assumption of perfect correlation also ensures that the network provisioned in Section 3.3 is “stress-tested”, i.e., made tolerant to a worst-case scenario where a chunk of capacity on correlated links is lost in one fell swoop.

Algorithm 1 Algorithm to generate timeseries of failures.

```

 $uf \leftarrow$  uptime fraction
 $ttr\_cdf \leftarrow$  CDF of TTRs from past data
 $tbf\_cdf \leftarrow$  CDF of TBFs from past data
 $timesteps \leftarrow$  no. of timesteps for which we generate timeseries

 $timeseries = 0$ 
Add 1 or 0 to  $timeseries$  with probability  $uf$  or  $1 - uf$  respectively
 $t = 1$ 
while  $t \leq timesteps$  do
  if last item in  $timeseries = 1$  then
     $tbf =$  Sample a value from  $tbf\_cdf$ 
    Add  $tbf$  1s to  $timeseries$  followed by a 0
  else
     $ttr =$  Sample a value from  $ttr\_cdf$ 
    Add  $ttr$  0s to  $timeseries$  followed by a 1
   $t = Length of timeseries$ 
Output: First  $timesteps$  elements of  $timeseries$ 

```

We use Algorithm 1 to generate a common link-status vector per-cluster. A value of 1 at $timeseries[t]$ denotes that all the links in the cluster are up at timestep t . Similarly, a value of 0 denotes that all the links in the cluster are in the failed state at timestep t . If the latest state of the generated timeseries is 1, we determine how much longer the links in the cluster will stay up by sampling a value, tbf , from the TBF distribution. Similarly, if the latest state is 0, we sample ttr from the estimated TTR distribution and fail all links in the cluster for the next ttr timesteps.

3.4.3 Properties of generated timeseries

We inspected some properties of the generated link failure timeseries and compare it with those of the actual failures recorded in the history.

Total network capacity available in a region at a given time is the sum of capacities of links that are up at that time. Therefore the total network capacity varies with time. Figure 3 shows the CDF and Box plots of the distribution of total network capacity in a region and compares the distributions that we see in the past data and the generated failure scenarios. 10 failure scenarios each spanning a month were generated from the past 13 months of logged data. The capacities are min-max normalized using the same minimum and maximum normalization factors for both generated and logged capacities. The generated timeseries has a distribution of total capacity that is reasonably close to the distribution we see in the past data, but does not exactly replicate it. In particular, the minimum generated capacity is lower than the minimum logged capacity. This is because, as noted above, when failure scenarios are generated, all links in a cluster are failed together, thereby generating worst case failure scenarios that may not have occurred in the past. This ensures that enough capacity is provisioned for scenarios not present in our logs but are nevertheless possibilities because of failure correlations and so would be prudent to be prepared for.

3.5 Provisioning and Control Interfaces

In this section, we describe the APIs required in the provisioning and control plane between the applications and the network to realize the gains from cooperative provisioning.

Every few months, each application component provides its immediate and deferrable demands between every DC pair as a time-series to the network. Figure 4 provides an example of how applications specify their demands to the network provisioning process. Immediate demands specify an hourly rate r while deferrable demands specify a volume V of traffic that is generated within each day. Additionally, for each application, the deferrable demand includes a deadline d and a probability of satisfaction p .

The diagram illustrates the specification of demands for two application components: Mail Client and Search. It shows two tables: Immediate Demands and Deferrable Demands.

Immediate Demands:

Time	From	To	Application	Rate (Gbps)
1-Jun-21 12:00:00	SIN	HKG	Mail Client	3000
1-Jun-21 12:00:00	SIN	HKG	Search	1000
1-Jun-21 13:00:00	SIN	HKG	Mail Client	2500

Deferrable Demands:

Time	From	To	Application	Volume (Tb)
1-Jun-21 12:00:00	SIN	HKG	Mailbox Load Balance	1500
1-Jun-21 12:00:00	SIN	HKG	Server Initialization	300
1-Jun-21 12:00:00	SIN	HKG	Periodic Tasks	50
2-Jun-21 12:00:00	SIN	HKG	Mailbox Load Balance	1200

Application	d	p
Mailbox Load Balance	7days	0.999
Server Initialization	7days	0.99

Figure 4: The specification of immediate demands using rate and deferrable demands using volume.

Choosing deadlines: In some cases, component owners can systematically calculate deadlines, while in others, it is left to domain experts to use their judgement and specify the deadline. Section 4.1 provides examples from the Substrate service that fall into both categories. To gain more insight into the choice of deadlines, we interviewed the service owners,

who are the domain experts. Since the service owners are going from operating in a mode without any deferrable traffic to one where they are willing to defer part of their traffic to enable cost savings, it is understandable that they were conservative in picking appropriate deadline values. Also, in some cases, they prefer the component to have a minimum amount of capacity available at all times. To support this, we simply divide the component's WAN traffic demands into an immediate component and a deferrable component. For instance, for the mailbox load balancing component described in Section 4.1, the domain experts require 10% of traffic to be flagged as immediate.

Forecasting: The network's provisioning process takes these inputs and forecasts future usage. To do this for immediate demands, it first aggregates hourly usage across all applications, and then computes the P95 usage over an extended period such as a day. Using this number across multiple days, it then determines a trend and forecasts future usage. For deferrable demands, the network forecasts daily volume of traffic for an extended period, say months, and inputs these to Approv. Appendix A.1 explains how Approv uses this input.

Run-time Interface: Since cooperative provisioning is done assuming that part of the demand is deferrable, cooperation of both the network and the application at run-time becomes necessary to live up to this assumption. Specifically, the application and the network need to react quickly and appropriately to link failure events. When a link fails, the network informs deferrable application components to either slow their rate of generating traffic or to pause such traffic altogether. We achieve this through a combination of two techniques: one enforced by the network and the other effected through application cooperation.

When traffic generated by a deferrable component is uniquely identifiable through network-visible identifiers, such as IP addresses or five-tuples, the network uses a bandwidth enforcer similar to previous work [12, 14] to apply back-pressure on the application. When it detects a link failure, the bandwidth enforcer reduces the allocation to the deferrable component that needs to be slowed down. Substrate's deferrable components (such as load-balancing, database migration, and background cleanup processes) are designed to slow down when the bandwidth enforcer reduces their allocation. As a result the components either reduce the amount of work they do or pause it altogether. When the failed link comes back up, the bandwidth enforcer increases the allocated bandwidth and sends an explicit signal to the application components to resume normal activity.

Often, however, multiple application components use the same network identifier to send traffic and consequently, network bandwidth enforcement cannot isolate traffic from the individual application components to be slowed down or paused. Fortunately, Substrate uses a job scheduler that controls the progress of background and asynchronous tasks. Hence when

the network informs the application of a link failure, the application’s job scheduler explicitly slows the appropriate deferrable component(s), thereby keeping the application’s network traffic in line with the reduced network capacity.

We are in the process of productizing WAN capacity provisioning at Microsoft based on *Approv*. This work is in collaboration with both the WAN team and the owners of the Substrate service. The implementation comprises *Approv*-based optimization of the offline provisioning pipeline and run-time adaptation. The latter comprises two parts. The first is bandwidth throttling enforced by the network, when there is loss of capacity due to link failure, to limit the traffic of applications that had marked part of their demand as deferrable during the offline provisioning phase. This ensures the protection of the network from overload during such times of capacity crunch, regardless of application actions. The second is run-time adaptation of the Substrate service, to ensure that the application throttles its activity, and hence traffic, in a manner that avoids user impact, e.g., by pausing deferrable components but not the user-facing ones. For this purpose, we have implemented interface enhancements using approximately 2500 lines of C# code, and are running comprehensive tests to verify that all Substrate’s deferrable components are indeed able to pause or slow down when required.

4 Evaluation

In this section, we evaluate the benefits of cooperative provisioning. Our evaluation uses Microsoft inter-DC WAN topology and link failure characteristics, and applies *Approv* to satisfy demands for Substrate, a large-scale service that supports several collaboration applications such as email, shared file services, and enterprise analytics. Substrate accounts for well over a third of the overall traffic on Microsoft WAN and as such plays a significant role in defining the WAN capacity. Therefore, we believe it is useful to analyze capacity even just in the context of the Substrate service. We first describe our methodology, and then turn to our results.

4.1 Application Demands

Substrate consists of various *components* which perform different logical functions. Many components are user-facing; for instance, a component that responds to a user’s REST API calls to read email. These components create immediate traffic demands. Additionally, Substrate implements a number of components that use the WAN extensively to ensure high data availability, reliability, and performance. These mostly run asynchronously and therefore their traffic demands are mostly deferrable. By interviewing component owners, we have determined the following four components whose application demands can be deferred.

Mailbox Load Balancing (MLB): This component is specific to the email application built on Substrate. With time,

utilization on some servers (measured in terms of CPU usage, storage or IO capacity) can become disproportionately high. To preempt this from impacting user latency, the load-balancing component periodically schedules mailbox moves from heavily loaded to lightly loaded servers. Such moves can be deferred until a certain deadline. To determine the right deadline, the load-balancing team continuously monitors the free and available resources (i.e., the “headroom”) on each server, e.g., the free storage available. Based on the rate at which utilization grows on each server and the available headroom on the various resources, the team calculates a deadline by which the load balancing must complete while still keeping the utilization under control and the user latency unaffected. Currently, this deadline is conservatively calculated as 7 days, which allows for occasional unexpected surges in load.

Periodic Tasks (PT): Substrate requires certain maintenance and analytics tasks to run periodically at a daily, weekly, or monthly cadence. These run in the background and perform two main functions. First, they perform data clean-up. For example, one task permanently deletes data items that are marked as deleted. Another task ensures compliance by performing time-driven deletions as mandated by legislation such as GDPR [2]. Second, they perform analytics tasks that extract useful enterprise-specific information from the data, such as generating weekly reports on how an employee splits their time between meetings and focus time. Based on conversations with owners of these tasks, we determined that these demands are deferrable by up to 1 day.

Server Initialization (SI): Substrate is a rapidly growing cloud service. To support this growth, it periodically adds new servers to its datacenters. This growth happens in bursts and is not gradual: a large number of new servers may become available at a particular datacenter in a particular month, and the next set of new servers may land only much later. The new servers are initialized with data from existing servers that may be in other datacenters, triggering large data transfers over the WAN. This workload is deferrable since such initialization does not have immediate, user-visible consequences. On the flip side however, it has to complete within a reasonable time so that the service can start utilizing the new servers, and thereby support growing demand. Per the operations team, 7 days is a reasonable deadline for the demand arising from the (new) server initialization. Although SI-based demands occur only sporadically, the network provisioning has to explicitly consider this demand since the traffic is bursty with a high peak.

Database Geo-Replication (DG): Substrate uses database-level geo-replication to ensure data availability. Changes to any data item are written to a database, and the database transaction log is replayed at three geo-replicated servers. Hence all components that modify data items generate geo-replication traffic. Replication traffic triggered by user-visible updates such as the creation of a new email is treated as immediate, while the rest is treated as deferrable, with a deadline

that is the same as for the component that generates it: 7 days for MLB and SI, and 1 day for PT.

4.2 Evaluation Methodology

We first describe the inputs used to evaluate Approv. Then, we outline the different provisioning approaches we compare Approv with. Finally, we explain the network topology that we provision for and the link failure characteristics Approv uses to simulate failures.

4.2.1 Input Demands

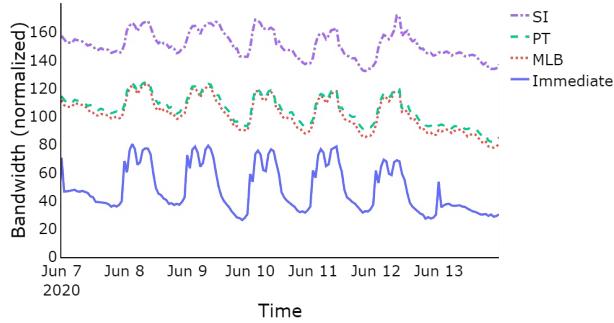


Figure 5: Substrate’s network usage over a week in June 2020 on a link from Singapore to Hong Kong, separated into immediate and deferrable (MLB, PT and SI).

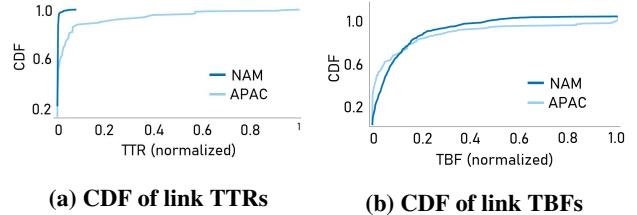
We use one month of network usage data from June 2020 to evaluate Approv. The deferrable demand constitutes 59% of total demand by volume, of which mailbox load balancing accounts for about 36%, periodic tasks for about 3% and server initialization for about 20% (see below for elaboration on SI traffic). Figure 5 shows immediate and deferrable demands (stacked over immediate) from a DC in Singapore (SIN) to a DC in Hong Kong (HKG) over one week in June 2020. The curve for each component includes both the traffic arising from updates to the primary replica and the database geo-replication triggered from updates to the secondary replicas.

We did not observe server initialization traffic in June 2020, which is the month for which we have detailed traffic traces. However, to determine the effect it would have on provisioning, we took an estimate of such sporadic traffic, obtained from the concerned team, and overlaid it on top of the total June 2020 traffic. The aggregate traffic traces so synthesized are then provided as input to the provisioning algorithm.

All demands are specified as a time-series in the format shown in Figure 4. Due to the commercial sensitivity of some of this data, as was alluded to in Section 1, we do not spell out the actual values here.

4.2.2 Provisioning Approaches

We quantify the benefits of Approv by comparing it with two alternative approaches:



(a) CDF of link TTRs (b) CDF of link TBFs

Figure 6: WAN link reliability statistics.

1. Baseline (BL): This is the current state-of-the-art provisioning approach mentioned in Section 3.1. There are several similarities between this baseline approach and the capacity planning scenario described in previous work [1, 21]. All application demands are treated equally. The demands are expressed as rates that need to be satisfied, and are derived by observing the P95 value of rate over the entire dataset. Failures are simulated as point-in-time events during which the demands are to be satisfied, rather than as failure episodes that the demand could tide over.

2. Smoothing-only (SO): This approach performs provisioning with differentiated application demands, but only for smoothing of the deferrable demand (expressed as a rate) to fit in the valleys of the immediate demand (see Figure 2). In this approach, once the deferrable traffic is smoothed into the valleys, we determine the P95 rate over the dataset and perform provisioning using point-in-time failure simulation. This approach allows us to evaluate how just smoothing traffic, which is well studied in the context of traffic engineering [7, 13], can help improve capacity provisioning.

4.2.3 Network Topology and Link Characteristics

Each provisioning approach mentioned above includes failure simulation. Our failure simulation method is driven by the network topology and link failure data from Microsoft WAN. We concentrate on two regions, Asia-Pacific (APAC) and North America (NAM), since they represent two very diverse network topologies. NAM is the largest region in Microsoft WAN in terms of both the number of links and capacity (it has tens of datacenters and hundreds of links), but consists of mostly land-bound links. APAC, on the other hand, while being smaller, includes an extremely diverse set of links, with a combination of land-bound links and several under-sea cables, given the geography of the region. Figure 6 shows the difference in link failure characteristics between APAC and NAM. While the TTR is almost uniformly low for the links in NAM, there is wide variation in the TTR for APAC, because some links (e.g., those on undersea cables) take time to repair. The TBFs are fairly similar across both regions.

We evaluate each provisioning approach with two failure simulation methods:

1. Replay-based: We simulate failures by replaying the 16-month link-status vector history from Microsoft available to us, with each month treated as a separate failure scenario,

Failure Method	Region	SO (% savings)			Approv (% savings)		
		MLB	+PT	+SI	MLB	+PT	+SI
Replay	APAC	6.5	6.5	9.7	16.8	17.4	38.1
	NAM	9.8	10.4	10.8	28.0	29.2	30.5
Generated	APAC	5.3	5.3	8.2	24.9	25.2	44.2
	NAM	10.8	10.9	11.1	27.3	29.0	31.8

Table 2: Percentage capacity savings over Baseline for SO and Approv, for both replay and failure generation. We first show savings with only Mailbox Load Balancing (MLB) considered deferrable, then we add on Periodic Tasks (PT). Finally, we add Server Initialization (SI) traffic as estimated by the SI team (shaded columns).

yielding a total of 16 scenarios for each of APAC and NAM. *2. Model-generated:* We use the failure model described in Section 3.4 to generate a synthetic but realistic set of link-status vectors.

While the replay-based approach enables evaluation independent of our failure generation model, the latter enables the creation of an unbounded number of failure scenarios (we create 10,000 for each of APAC and NAM), in turn allowing us to evaluate the impact of varying the probability of satisfaction, p (from 0.9 to 0.999).

4.3 Results

In this section, we first evaluate how Approv reduces the provisioned network capacity while supporting Substrate’s demand, and improves link utilization compared to the baseline. Next, we perform a sensitivity analysis to evaluate how Approv’s savings vary with d and p . Finally, we examine and compare forecasting for the baseline and Approv.

Network Capacity Reduction²: We first evaluate the percentage reduction in network capacity provisioned using the three approaches. We calculate the network capacity as the sum of all link capacities in the network. Table 2 shows the percentage network capacity reduction for the Asia-Pacific and North America regions, for both methods of simulating failures: replay-based and model-generated. As noted in Section 4.2.1, since the Substrate components that write to data items trigger database geo-replication (DG), we have included the DG demands in those of the respective Substrate components. Since the historical time-series contains 16 months of data, for the generated method, we generate link-status time-series of length 16 months too.

We concentrate on the last row of the table (NAM with failure generation), though the same explanation applies to the other rows as well. Approv, with MLB alone treated as deferrable, reduces provisioned capacity by 27.3%. With the deferrable set expanded to also include PT, which is a smaller workload and with a tighter deadline of only 1 day, the gains

²In general, we would also want to consider link-cost-weighted capacity savings, but we defer this to future work.

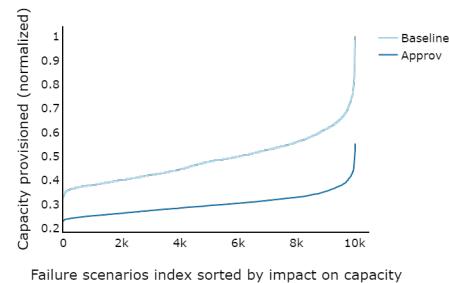


Figure 7: Incremental augments in network capacity for the APAC region for Baseline and Approv.

increase to 29%. Though it is a small increase, there is still value in considering demands like PT as deferrable, since even a small improvement in percentage capacity provisioned yields large absolute gains. Finally, the inclusion of SI as well in the deferrable set adds to the gains, taking it up to 31.8%. The benefits of Approv over SO are also apparent, since SO gives gains of 11.1% whereas Approv’s total gains are much higher at 31.8%. Note that both failure simulation methods yield a significant reduction in the capacity to be provisioned, underscoring that the gains are not just specific to our failure model.

Including SI in the deferrable set provided significantly higher gains in APAC than NAM (e.g., an increase of about 20% in capacity savings in APAC compared to just 1-3% in NAM). This is because the number of servers added to APAC and NAM datacenters, and hence the volume of initialization data, was roughly the same in both regions. Since Substrate service is much larger in NAM, the addition contributed less WAN traffic relative to the total in NAM compared to APAC.

Link Utilization Improvement: We now discuss the relative improvement in link utilization due to Approv. We analyze the utilization improvement in APAC network provisioned with Approv using the replay failure scenarios. The relative improvement is computed as $100 * (\text{Approv utilization} - \text{Baseline utilization}) / (\text{Baseline utilization})$ for each link. Since Approv reduces network capacity significantly, the link utilization improve significantly as well, with an average relative improvement of 42% (note that we compute the utilization improvement on a per-link basis and report the arithmetic average whereas the capacity savings is reported in aggregate).

Sensitivity Analysis: Figure 7 shows 10000 generated failure scenarios for the APAC region on the x-axis, sorted by the total network capacity that is provisioned for each failure scenario considered individually. We use all traffic (Immediate, MLB, PT and SI) in this experiment. The graph shows that Approv almost uniformly provisions about 30% less capacity than Baseline. This shows that irrespective of the exact nature of the failure scenarios, Approv consistently helps reduce the network capacity to be provisioned.

Experiment	$p=0.999$	$p=0.99$	$p=0.9$
Baseline	1.00	0.90	0.75
$d=12\text{hrs}$	0.66	0.61	0.51
$d=1\text{day}$	0.62	0.56	0.47
$d=2\text{days}$	0.61	0.51	0.43
$d=4\text{days}$	0.60	0.51	0.42
$d=7\text{days}$	0.60	0.51	0.42

Table 3: The sensitivity of capacity provisioned to the deadline d (varied across rows) and the probability of satisfaction p (expressed in terms of 9s and varied across columns). We normalize all values by the capacity provisioned for the Baseline approach with $p = 0.999$.

Next, we evaluate how Approv’s provisioned network capacity varies as we sweep through a range of settings for the deadline, d , and the probability of satisfaction, p . Table 3 shows the results in terms of the normalized capacity provisioned for the baseline case (the first row) and for Approv (the remaining rows).

We see that the capacity to be provisioned decreases as the probability of satisfaction p decreases and also as the deadline d increases. Both of these are as expected since the less “demanding” a demand is, the greater the opportunity Approv has to accommodate the demand without increasing the capacity to be provisioned. However, we also see that the capacity tends to flatten out as the deadline increases, with a diminishing benefit to relaxing the deadline. The reason is that Approv is able to effectively utilize the large amount of headroom in the network arising from the redundant capacity provisioned to support the immediate demand. Note that in the absence of failures, this redundant capacity is not needed for serving the immediate demand and so is available for Approv to accommodate the deferrable demand, even with a shorter deadline.

Table 3 also points to an interesting tradeoff between p and d . For instance, the normalized capacity (0.61) with $p = 0.99$ and $d = 12\text{hrs}$ is roughly the same as it is (0.6) when p is made more demanding ($p = 0.999$) but d is relaxed to $d = 7\text{days}$.

Forecast: Finally, we evaluate the effect of forecasting error on Approv. We used 3 months of traffic to forecast 10 days of traffic using the Holt-Winter method of forecasting [6] with a seasonality of 7 days. To ensure that the network is able to accommodate sudden, unforeseen peaks in traffic, network forecasts typically use a high confidence interval (CI) envelope, hence, we use a confidence interval of 95% in our experiment. First, we forecast overall traffic, without differentiating immediate and deferrable traffic, as is the current state of the art. Next, we forecast immediate and deferrable traffic separately, as is required by Approv.

For overall demand, the forecast overestimation error for the high end of the 95% CI is 12%. When we separately forecast the deferrable demand, the forecast overestimation error is 11%. In other words, the forecasting is about as accurate for the deferrable part of the traffic as it is for the overall traffic.

Therefore, Approv could as well use the forecast demand as input as it can the current snapshot of the demand (as we do in this paper, using the June 2020 snapshot). Furthermore, given the large amount of headroom that is created by provisioning of immediate traffic (shown in Table 3), we believe Approv has an overall low sensitivity to forecasting errors as well. For these reasons, we believe Approv will remain effective even when used with demand forecasts.

5 Discussion

In this paper, we have used the applications’ deadline and desired probability of satisfaction information to provision the network optimally. One challenge that arises from this is incentivizing applications to specify these requirements, which convey their flexibility, appropriately. If the applications are too conservative in specifying their requirements, that would result in reduced capacity savings, whereas if the applications specify their demands loosely, then that might result in an incorrect forecast and hence inadequate network provisioning. We believe that this opens up the possibility of devising pricing mechanisms to encourage applications to specify their demands appropriately, thereby facilitating the cooperative provisioning of network capacity. Another challenge is in extending cooperative provisioning to third-party applications. Unlike with first-party applications, where we assume an implicit trust between the applications and the network that enables free sharing of information up and down the stack, in the third-party setting, we would need to rethink this approach and consider how cooperation can be effected in the absence of such trust.

6 Related Work

In this section, we position our work in relation to previous efforts in traffic engineering, network provisioning and failure characterization.

Traffic Engineering: As discussed in Section 1.1, traffic engineering employs routing and scheduling strategies to satisfy the possibly differentiated demands of applications on a given network. B4 [8] and SWAN [7] use routing algorithms that allow flows to specify different priority levels. Tempus [11] and Amoeba [22] allow applications to set deadlines for traffic, the former uses constraint optimization to satisfy deadlines while the latter uses a graph-based algorithm. Failure-aware traffic engineering has also been explored. FFC [18] splits traffic over multiple paths to be resilient to failures. Song et al. [17] proposed an availability-aware traffic engineering algorithm for optical networks that we believe can be generalized to WANs. NetStitcher [13] uses store-and-forward techniques to deal with temporal offsets between traffic peaks in different parts of the network.

Pretium [10] uses dynamic pricing to route third-party application traffic while handling link failures.

It is important to recognize that traffic engineering and network provisioning are fundamentally different problems. Traffic engineering addresses the short-term, i.e., run-time, problem of how networks route traffic while assuming given link capacities provided as input. Provisioning, on the other hand, addresses the longer-term problem of determining how link capacities should be provisioned in the first place, possibly several months or even over a year into the future, so as to satisfy the communication requirements of applications. In doing so, the provisioning framework (such as Approv in this paper) would invoke traffic engineering techniques under the hood and iteratively, to check if the demand can be satisfied by the network as provisioned and in the face of one or more failure scenarios. If traffic engineering is unable to route the demand, the capacity provisioned would be augmented.

Network Provisioning: We now discuss previous work that directly addresses the problem of network provisioning. Robust network validation [1] proposes a generic optimization framework to determine worst-case network performance across multiple scenarios. They use this framework to model the network provisioning problem which determines the right augments to link capacity so as to handle multiple failures. Liu et al. [21] propose an optimization framework that also solves the network provisioning problem given a set of failure scenarios. Both efforts address network provisioning, however, without using first-party context, and therefore consider all application traffic to be equivalent. Moreover, since they do not inherently consider deferrable demands, they use a simple failure model that only simulates instantaneous, point-in-time failures, with no notion of the temporal dynamics of link failures, i.e., a link failing at a certain time but then being restored at a later time. The baseline provisioning approach we have used in Section 4 is derived from these techniques.

Failure Characterisation: Turner et al. explore and characterize network failures in the CENIC network and provide a methodology to reconstruct these failures [20]. This is similar to the replay-based model we use in Section 4. Shaikh et al. [16] similarly measure link status over time using OSPF link-state advertisements in a large enterprise network. Gill et al. [5] characterize network failures within a datacenter, not on the WAN. Unlike our failure modeling methodology, these efforts analyze historical failures and do not propose generative failure models based on this history. Previous work [4] analyzes optical layer outages in a large backbone and shows that Q-drop events are predictive of upcoming failures. We believe that augmenting our failure model with information from the optical layer would be an interesting future direction.

7 Conclusion

We have described how cooperation, cutting across applications and the network, can significantly lower capacity pro-

visioned in inter-DC WANs in a first-party setting. Using co-operative provisioning, we show how we can provision the WAN more carefully while ensuring that application demands are accommodated with the desired satisfaction probability.

Acknowledgements. We thank our shepherd, John Wilkes, and the anonymous reviewers for their feedback. We are particularly grateful to John for his meticulous shepherding of our paper. We also thank Neha Agrawal, Matt Calder, Wengjie Chen, Christina Chou, Pavel Egorov, Luis Irún-Briz, Jim Kleewein, Angus Leeming, Shijuan Lu, Otávio Pereira, Saravanakumar Rajmohan, Nadia Razek, Xiaoshi Sha, Jianke Tang, Jiaojian Wang, and Paul Wang for their help and input during the course of this work.

References

- [1] Y. Chang, S. Rao, and M. Tawarmalani. Robust validation of network designs under uncertain demands and failures. In *USENIX NSDI*, pages 347–362, 2017.
- [2] European Commission. 2018 reform of EU data protection rules. https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf. Accessed Sep 12, 2021.
- [3] B. S. Everitt, S. Landau, M. Leese, and D. Stahl. *Cluster analysis 5th Edition*. John Wiley, 2011.
- [4] M. Ghobadi and R. Mahajan. Optical layer failures in a large backbone. In *ACM IMC*, page 461–467, 2016.
- [5] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM*, 2011.
- [6] C. C. Holt. Forecasting seasonals and trends by exponentially weighted moving averages. *International Journal of Forecasting*, 20(1):5–10, 2004.
- [7] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, page 15–26, 2013.
- [8] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, K. N. B., C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev, S. Padgett, F. Rabe, S. Ray, M. Tewari, M. Tierney, M. Zahn, J. Zolla, J. Ong, and A. Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google’s software-defined WAN. In *ACM SIGCOMM*, page 74–87, 2018.
- [9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu,

- J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, page 3–14, 2013.
- [10] V. Jalaparti, I. Bliznets, S. Kandula, B. Lucier, and I. Menache. Dynamic pricing and traffic engineering for timely inter-datacenter transfers. In *ACM SIGCOMM*, pages 73–86, 2016.
- [11] S. Kandula, I. Menache, R. Schwartz, and S. R. Babu. Calendaring for wide area networks. In *ACM SIGCOMM*, pages 515–526, 2014.
- [12] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasi-nadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *ACM SIGCOMM*, page 1–14, 2015.
- [13] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez. Inter-datacenter bulk transfers with netstitcher. In *ACM SIGCOMM*, page 74–85, 2011.
- [14] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *ACM SIGCOMM*, page 188–201, 2016.
- [15] D. Salinas, V. Flunkert, J. Gasthaus, and T. Januschowski. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 36(3):1181–1191, 2020.
- [16] A. Shaikh, C. Isett, A. Greenberg, M. Roughan, and J. Gottlieb. A case study of OSPF behavior in a large enterprise network. In *ACM SIGCOMM Workshop on Internet Measurement*, page 217–230, 2002.
- [17] L. Song, J. Zhang, and B. Mukherjee. Dynamic provisioning with availability guarantee for differentiated services in survivable mesh networks. *IEEE Journal on Selected Areas in Communications*, 25(3):35–43, 2007.
- [18] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford. Network architecture for joint failure recovery and traffic engineering. *SIGMETRICS Perform. Eval. Rev.*, 39(1):97–108, June 2011.
- [19] S. Taylor and B. Letham. Forecasting at scale. PeerJ Preprints 5:e3190v2 <https://doi.org/10.7287/peerj.preprints.3190v2>, 2020. [Accessed March 10, 2021].
- [20] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: Understanding the causes and impact of network failures. In *ACM SIGCOMM*, 2010.
- [21] Yu Liu, D. Tipper, and P. Siripongwutikorn. Approximating optimal spare capacity allocation by successive survivable routing. In *IEEE INFOCOM 2001*, pages 699–708 vol.2, 2001.
- [22] H. Zhang, K. Chen, W. Bai, D. Han, C. Tian, H. Wang, H. Guan, and M. Zhang. Guaranteeing deadlines for inter-data center transfers. *IEEE/ACM Transactions on Networking*, 25(1):579–595, 2016.

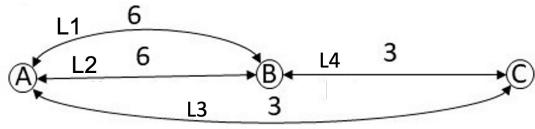


Figure 8: Toy Example

A Appendix

A.1 LP formulation

Objective Function: Minimize weighted sum of capacity augments where weight could be link cost, latency, etc.

$$\text{Minimize} \sum_l w_l \cdot X_l \quad (1)$$

Subject to the constraints:

A. Demand constraints at sources and destinations:

$$\sum_{t_i \leq t \leq t_i + d_i} \sum_{l \in OL_{S_i} \cap L_{R_i}} v_{t,l}^{f,i} = V_i \quad \forall f \in F, i \in D \quad (2)$$

$$\sum_{t_i \leq t \leq t_i + d_i} \sum_{l \in OL_{T_i} \cap L_{R_i}} v_{t,l}^{f,i} = V_i \quad \forall f \in F, i \in D \quad (3)$$

Constraint (2) ensures that for any deferrable demand, i , the total volume of traffic emerging from its source node, S_i , over all time steps within its deadline is equal to the actual demand volume V_i . Constraint (3), on the other hand, ensures that the total volume traffic arriving at the destination node over all time steps within its deadline is also equal to actual demand volume V_i . In computing the total volume in constraint (2), we only consider the subset of links emanating from the source, S_i , that also lie on a valid path between S_i and the destination, T_i , and likewise in constraint (3).

B. Network flow constraint at each node, n :

$$\sum_{i \in D} \sum_{l \in IL_n} v_{t,l}^{f,i} + \sum_{\forall i | S_i = n} V_i = \sum_{i \in D} \sum_{l \in OL_n} v_{t,l}^{f,i} + \sum_{\forall i | T_i = n} V_i \quad \forall f \in F, n \in N, t \in T \quad (4)$$

This is a network flow constraint to ensure that, in each time slice, the sum of the volume coming into the node and volume generated at that node is equal to the sum of volume going out of that node and volume sinking into that node.

C. Capacity constraint at each link, l :

$$(E_l + X_l) \cdot u_{t,l} \geq I_{t,l}^f + \sum_{i \in D} v_{t,l}^{f,i} \quad \forall l \in L, f \in F, t \in T \quad (5)$$

This constraint ensures that the total volume of traffic that a link is able to carry during a time step (computed as a product of the total capacity provisioned on that link to accommodate the immediate and deferrable demands and the uptime of the link) should be greater than or equal to the sum of immediate and deferrable demands volume routed via that link during that time step.

Note that all of the above constraints apply during each failure scenario, f , so there is a set of such constraints corresponding to each such failure scenario.

A.2 Example of Cooperative Provisioning

We use a toy example network (Figure 8) with 3 datacenters and 2 immediate and 2 deferrable demands to illustrate steps 1 and 2 from Section 3.3. Table 4 shows the TTR and TBF of each link. For fair comparison with the baseline, we simulate 2-link simultaneous failures, in which links L1 and L3, and links L2 and L4 can fail together (the high TTR of 15 days for link L3 makes it more likely that its failures would overlap with those of the other links), along with all single link failures. Table 5 shows the input in terms of the immediate and deferrable traffic demands. Immediate demands R1 and R2 require a peak rate of 3 GB/day but sends 2 GB/day on average, hence sending total 60 GB in a period of 30 days.

1. Baseline :

Table 6 shows point-in-time failure scenarios simulated as part of the baseline provisioning described in the Section 4. Consider the point-in-time failure scenario 2 in Table 6, where links L1 and L3 fail together, hence the required rate between A and B is 6 GB/day to satisfy demands R1 and R3 and required rate between A and C is 9 GB/day to satisfy demands R2 and R4. And as L2 and L4 constitute the only available path, we need to allocate at least 15 GB/day capacity on L2 and at least 9 GB/day capacity on L4, hence capacity augments of 9 GB/day and 6 GB/day are required on L2 and L4, respectively. Provisioning for all such failure scenarios results in a total augment of 30 GB/day. With all demands expressed as peak rates that must be satisfied even when there are failures, point-in-time failure scenarios are effectively equivalent to a complete failure of a link over the entire 30-day period. Such a conservative approach is unnecessary for deferrable demands provisioning, as discussed next.

2. Approval:

Step I : We do peak-rate based point-in-time failure scenario provisioning for immediate demands which is similar to baseline. Considering the same point-in-time failure scenarios as in the baseline provisioning, we find that the topology in the

Failure Scenario	X1	X2	X3	X4
No failure	0	0	3	0
L1 and L3 fails	0	9	0	6
L2 and L3 fails	9	0	0	6
L4 fails	0	0	3	0
Final Augments	9	9	6	6

Table 6: Baseline provisioning with point-in-time failure scenarios

Failure Scenario	X1	X2	X3	X4
No failure	0	0	3	2
L1: [15, 17], L3:[15, 30]	0	0	0	5
L2: [15, 20], L3:[15,30]	0	0	0	5
L4: [15, 20]	0	0	5	0
Final Augments	0	0	1.67	5

Table 7: Volume based LP provisioning with timeseries failure scenarios

Link	TTR	TBF
L1	2	30
L2	5	60
L3	15	30
L4	5	60

Table 4: Link TTRs

Figure 8 is enough to satisfy the immediate demands, R1 and R2.

Step II: In this step, we provision for deferrable demands R3 and R4 on top of the Step 1 topology, while working with time series failure scenarios, as discussed in Section 3.4. Note that we have only shown a few failure scenarios (which cause the highest augments) due to lack of space but the final augments are based on simulating plenty of such failure scenarios.

Consider the failure scenario 2 in which L1 fails at day 15 and comes back up at day 17 and L3 fails at day 15 and comes back up at day 30. We don't need to provision extra capacity for demand R3 because if we send more volume of R3 between day 0 and day 15, we would not need to send any traffic between day 15 and day 17. On the other hand, since the demand R4 starts at day 15 and we need to send 90 GB in a period of 15 days, which coincides with the downtime of link L3, we would need to provision capacity on an alternate path to accommodate demand R4 in the face of such a link failure. Specifically, if we consider link L4 on the alternative

path, 15 GB out of the total R4 demand of 90 GB could be accommodated in the valleys of immediate demand R2. That would leave 75 GB of R4 to be satisfied, necessitating $75/15 = 5$ GB/day of extra capacity on link L4 to satisfy deferrable demand R4, alongside immediate demand R2 flowing over the same link, L4. Note that since we solve one LP optimally across all failure scenarios, the final capacity augments yielded by our LP is not necessarily the same as the maximum capacity augment required on each link across the individual failure scenarios.

Request	Type	Peak Rate	Start Time	Deadline	Volume
R1: A->B	Immediate	3 GB/day	-	-	60 GB (30 days)
R2: A->C	Immediate	3 GB/day	-	-	60 GB (30 days)
R3: A->B	Deferrable	-	0th day	30 days	90 GB
R4: A->C	Deferrable	-	15th day	15 days	90 GB

Table 5: Immediate and Deferrable demands

Some observations on Approv provisioning:

1. Baseline provisioning requires 30 GB/day total capacity augments whereas Approv requires only 6.67 GB/day total capacity augments.
2. For temporally overlapping demands, Approv will find optimal allocation. For example, deferrable demand R3 will send more bytes in the first 15 days to utilize the network well and will leave enough capacity on links L1 and L2 for sending R4 traffic between day 15 and day 30.
3. In our example, deferrable demands could be routed via either link L3 or link L4, but Approv returns higher augments on link L4 because it has lower TTR value. In effect, Approv favors augmenting low TTR and high TBF links, i.e., high-availability links.
4. As our objective function is to minimize total weighted capacity augments, Approv favors augmenting shorter paths.
5. With our volume-based LP formulation, we do not need to explicitly smooth our input demands, since Approv fills deferrable traffic in the immediate traffic valleys implicitly as part of its optimization.

OrbWeaver: Using IDLE Cycles in Programmable Networks for Opportunistic Coordination

Liangcheng Yu

University of Pennsylvania
leoyu@seas.upenn.edu

John Sonchack

Princeton University
jsonch@princeton.edu

Vincent Liu

University of Pennsylvania
liuv@seas.upenn.edu

Abstract

Network architects are frequently presented with a tradeoff: either (a) introduce a new or improved control-/management-plane application that boosts overall performance, or (b) use the bandwidth it would have occupied to deliver user traffic.

In this paper, we present OrbWeaver, a framework that can exploit unused network bandwidth for in-network coordination. Using real hardware, we demonstrate that OrbWeaver can harvest this bandwidth (1) with little-to-no impact on the bandwidth/latency of user packets and (2) while providing guarantees on the interarrival time of the injected traffic. Through an exploration of three example use cases, we show that this opportunistic coordination abstraction is sufficient to approximate recently proposed systems without any of their associated bandwidth overheads.

1 Introduction

The purpose of a computer network is to transmit messages to and from connected devices. The bulk of these messages are sent between two or more end hosts and are intended for use in applications therein (video streaming, web browsing, ssh terminals, stock trackers, etc). It is important to note, however, that networks are also frequently used for other purposes that are not directly related to end-to-end application traffic. These uses include but are not limited to control messages, keepalives, and probes.

In some cases, this second category of messages is sent over dedicated networks (e.g., an out-of-band control plane). Nevertheless, a significant portion is not, and for good reason. Multiplexing the traffic over a unified network results in more efficient resource utilization and helpful fate-sharing properties. For many uses, it is also required for correctness. For instance, active probing generally relies on the probe facing the same network conditions as normal traffic.

For in-band coordination, there is often a choice between fidelity and overhead. More so as many protocols use high-priority messages that directly cut into network capacity. For example, when deciding on an appropriate interval for sending routing-protocol keepalive messages, sending keepalives more frequently results in faster failure detection but at the cost of many extra packets in the network. Similarly, while techniques like congestion tagging [3, 22] and in-band network telemetry [27] can provide timely information about the

recent state of network paths, they require either extra probe packets or space in the headers of existing packets, both of which occupy valuable bandwidth.

Given this tradeoff between fidelity and overhead, today’s networks end up settling for a little bit of both. In some cases, the sacrifices are modest; in others, network operators are forced to limit the aggressiveness of their systems despite evidence of the benefits of finer granularity [6, 49]. In this paper, we argue that for a diverse set of protocols, the sacrifice is entirely unnecessary—systems can coordinate at high-fidelity with a near-zero cost to usable bandwidth and latency. In short: we can have our cake and eat it too.

Our system, OrbWeaver, is a framework for the opportunistic transmission of data across today’s programmable networks. OrbWeaver takes advantage of gaps between user traffic and ‘weaves’ (i.e., injects) into *every* such gap customizable IDLE packets that convey information across devices. For modern, high-speed networks, these opportunities are plentiful. Crucially, OrbWeaver provides guarantees about the ‘weaved’ stream—guarantees on the maximum time between any two packets and guarantees on the impact of the injected packets on user traffic, switch resources, and power draw. A consequence of this predictability is that, even when there is no opportunity to send, the absence of IDLE packets reveals concrete information about the state of the network.

We note that a similar abstraction already exists at the data-link layer. In particular, in today’s full-duplex Ethernet standards, the Physical Coding Sublayer (PCS) will fill any gaps in transmission with IDLE symbols [32, 41]. The continuous stream of incoming signals allows receivers to—with no impact to user traffic—test for corruption and link integrity at a fine granularity, even when there is no traffic on the network. Further, by continuing to transmit IDLE symbols after a link integrity issue has been raised, switches can also determine when the link becomes usable again.

OrbWeaver extends this technique to higher layers of the network stack by exploiting the data plane programmability, architecture, and packet generation capabilities of emerging programmable switching platforms. The resulting stream of packets can be used to generalize Ethernet’s robust failure detection properties to a broader class of faults; however, its benefits go far beyond L3 failure detection. Rather, we demonstrate in this paper that with proper application, the nearly free communication that IDLE packets provide can be used to eliminate the fidelity-utilization tradeoff of solutions

to several classic problems in networking including clock synchronization and load balancing.

Implementing OrbWeaver’s packet weaving presented several technical challenges. First, while IDLE symbols are part of the Ethernet standard and enjoy direct hardware/protocol support, to utilize today’s devices and maintain their current performance, OrbWeaver must provide similar behavior without changes to switch architectures. Second, while many systems can benefit directly from opportunistic data transmission, many must continue to operate during periods where user traffic is already occupying all available bandwidth. To address the first challenge, OrbWeaver introduces a co-design of the selective data-plane filtering mechanisms and the rich priority configurations found in modern switches to guarantee minimal impact on user traffic. We verify the approach through a detailed examination of the specifications of the queuing subsystems on a Tofino switch along with experiments that stress-test worst-case behavior. To address the second, we introduce novel mechanisms that exploit IDLE packets and the guarantees of weaved streams to eliminate the bandwidth overheads of existing network protocols. We demonstrate these mechanisms through three case studies.

Our implementation¹ and evaluation demonstrate the efficiency and efficacy of OrbWeaver using real hardware, optical attenuators, and power meters. We find that, despite the introduction of the IDLE stream, OrbWeaver incurs negligible impact on user traffic, the computational/state resources of participating switches, or their power draw. We further demonstrate that this messaging substrate can be used to (re-)design recently proposed systems to eliminate their bandwidth overheads while closely approximating their performance.

2 Motivating Weaved Streams

This section presents the definition of a ‘weaved’ stream, its motive, and where data plane programmability can help.

Definition. A *weaved stream* is a union of user and IDLE packets traversing a link between two arbitrary network devices that provides two guarantees:

- R1 That no link stays unutilized for too long. More precisely, there is some period τ where the interval between any two consecutive packets, d , satisfies $d \leq \tau$.
- R2 That the injected packets do not decrease the effective throughput of user traffic or increase their loss rate.

Why weaved streams? Network protocols are, fundamentally, distributed computations that require coordination between different devices—sometimes adjacent devices, sometimes remote devices—for monitoring, control, and management. A perennial problem is how much bandwidth to allocate to these protocols, as each byte devoted to coordination is a byte that could have been used for user traffic instead. This tradeoff has tangible effects for many networking tasks:

¹Code is available at <https://github.com/eniac/OrbWeaver>

- *Failure handling:* A common strategy for detecting the failure of remote network devices is the use of continuous keepalive messages. Here, each node periodically sends a keepalive to each of its neighbors; if a neighbor ever stops sending keepalives, nodes assume that they have failed. Fundamentally, the period between keepalives bounds the speed at which we can detect failures. Unfortunately, because keepalives are most accurate when sent over the same or higher-priority channels as user traffic, their sending rate is typically kept low (e.g., at a period of $O(100\text{ ms})$) at the cost of slower detection.
- *Clock synchronization:* Prior work has also noted the utility of synchronizing network devices [29], e.g., for coordinated network updates [36, 45] or real-time streams [13]. Clock synchronization protocols typically pass messages that periodically compute the drift between the clocks of participating machines. Constant changes to not only the relative clocks but also the relative clock *rates* mean that more frequent updates can provide more accurate synchronization (at the cost of additional packets in the network, typically configured at a high priority).

- *Congestion notification:* Finally, this tradeoff can be seen in the detection/communication of congestion and current load. ACKs (and their corresponding loss/RTTs) are a particularly common method for inferring the presence of congestion, e.g., in TCP NewReno. As others have noted [3, 26], however, there are also advantages to more explicit signaling of the current congestion and queue statistics. Unfortunately, while effective, these statistics typically occupy packet header space or introduce additional packets into the network.

Over the years, network architects have developed many workarounds. These include hardware changes [29, 32], co-opting unused fields in headers [3, 50], carefully balancing the tradeoff for a particular service-level expectation [7], or otherwise coming to terms with the cost of coordination. Outside factors can guide the above decisions, such as whether ACKs are already necessary (e.g., for reliability) or if extraneous fields can be eliminated. However, in this paper, we ask a more fundamental question: are these tradeoffs necessary?

To that end, OrbWeaver is a framework for implementing network coordination that does not interfere with user traffic. OrbWeaver’s weaved streams are both opportunistic and highly predictable—consuming every inter-packet gap of sufficient size but no more. Not every protocol can be implemented solely using weaved streams (though many can benefit from it). Even so, we demonstrate that at least for the three use cases above, weaved streams are sufficient to approximate state-of-the-art systems while reducing their impact on user traffic to virtually zero.

Why are there gaps? Usable gaps between packets can occur for many reasons, the most basic being application-level patterns and TCP effects. Indeed, prior work [37, 49] and

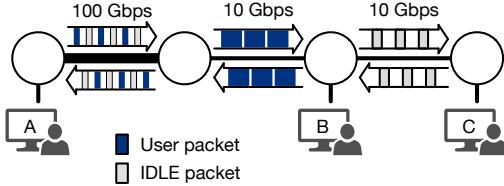


Figure 1: An example OrbWeaver-enabled network with four switches and three end hosts (connected with 10 Gbps links). A single two-sided connection $A \leftrightarrow B$ occupies the network, but a significant portion remains unused. Gaps between packets can occur for many reasons, but OrbWeaver can weave IDLE packets into all of those gaps.

our conversations with several large clouds/ISPs verify that micro-/milli-second inter-packet gaps are ubiquitous, even in networks that primarily handle large bulk-data transfers.

Gaps can also happen for structural reasons. For example, consider Figure 1 (sans IDLE packets). In it, a single connection $A \leftrightarrow B$ occupies all usable end-to-end bandwidth. Even if A and B pace packets perfectly, no host can send additional packets without displacing the existing user traffic, despite significant opportunities to do so (because of, e.g., congestion, link speed changes, and asymmetric connections). These gaps present a chance for opportunistic coordination.

Why now? OrbWeaver’s ability to weave IDLE packets into gaps between user traffic is enabled by several features in modern switches: programmable data plane behavior, the capacity for local packet generation, and the ability to fully configure the queuing/prioritization of different traffic classes. We note that none of these are sufficient on their own.

For example, consider strict packet prioritization, which has been used for opportunistic bandwidth allocation [21, 24]. In SWAN [21], for instance, end hosts send low-priority background traffic to capture any bandwidth remaining after handling interactive and elastic services. A naïve application of these techniques, however, is a poor fit for in-network coordination, which occurs between devices in the network (as opposed to end hosts) and typically involves small data sizes that benefit from even short sending opportunities. Figure 1, for example, would not benefit from end-host actions.

3 Generating a Weaved Stream

Before we delve into the potential uses of weaved streams in Section 4, we first detail how to implement the abstraction in today’s programmable switches.

Switch model. For simplicity, we primarily focus on the popular Tofino family of programmable networking devices (and discuss generalization to other types of devices in Appendix B). Figure 2 shows a conceptual diagram of the relevant components of the switches we consider. At a high level, when a packet enters from one of the Ethernet ports, its header is extracted by the programmable parser responsible

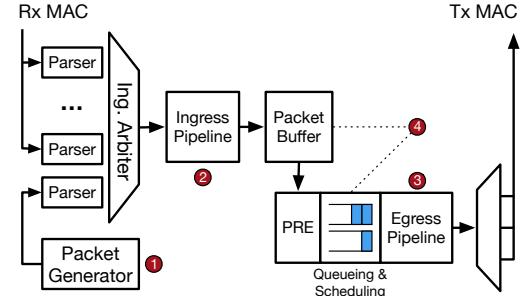


Figure 2: Conceptual diagram of the relevant components of an RMT switch, derived from the switch specifications in [12]. Only a single ingress/egress pipeline are shown. Circled numbers indicate steps and potential points of contention with user traffic that are handled in Section 3.1.

for that port. An ingress pipeline arbiter is then responsible for selecting one of the parsed packets and passing it through the ingress match-action pipeline.

After ingress processing, the packet will be placed in a shared packet buffer until it is ready to be sent out. Instead, the switch uses a shorter ‘packet descriptor’ for the next steps: optional replication by a Packet Replication Engine (PRE) (e.g., for multicast) and placement onto a per-port egress queue for eventual processing/deparsing. The data plane program and the traffic manager configuration decide whether an incoming packet should be buffered and whether a buffered packet should be enqueued for transmission.

Goal. R1 of the weaved stream abstraction requires a constant stream of packets on every link such that the union of user and IDLE packets satisfies $d \leq \tau$. We note that the optimal guarantee for τ is dependent on both the bandwidth, B , of each link and the MTU of the network. To see why, consider the extreme case where a user is occupying all of the bandwidth of a port i with MTU-sized packets. The receiver on the other side of the link will receive packets at a period of $\tau_i = \frac{\text{MTU}}{B_i}$, with OrbWeaver unable to inject any additional packets without impacting user traffic. Therefore, unless otherwise noted, OrbWeaver uses $\tau_i = \frac{\text{MTU}}{B_i}$ even if smaller IDLE packets would allow for faster injection.

In the worst case when there were zero user packets and N egress ports, the resulting target IDLE-injection rate is:

$$T = \sum_{i=1}^N \frac{B_i}{\text{MTU}}$$

For reference, for a 32 port switch with $B = 100$ Gbps and $\text{MTU} = 1500$ B, the per-port inter-packet gap, τ_i , is 120 ns, which results in $T = 266.7$ M packets/sec.

Constraints. Complicating the injection of IDLE packets into the network are R2 and hardware constraints on the throughput of each switch pipeline, defined in terms of both byte-level bandwidth ($N \times B$) and packet-level bandwidth (proportional to the clock rate of the pipelines). For the latter, switches

typically provide guarantees up to a certain minimum packet size, and best-effort behavior for very small packets.

3.1 Mechanism Overview

OrbWeaver’s IDLE-packet weaving leverages a combination of features found on our target platform: data-plane packet generation, data plane programmability, and fine-grained arbiter/scheduler configuration options. The switches’ onboard per-pipeline packet generator modules, in particular, form a convenient substrate for our techniques. Using these modules, a network operator can create packets with predetermined content at a predetermined rate.

In principle, one could configure the generators to create packets at a rate T (thus providing OrbWeaver with its consistent stream of packets to convert into IDLE packets). Unfortunately, in practice, these generators do not have nearly enough capacity to satisfy the requirements of OrbWeaver. Moreover, blind injection of packets may interfere with the throughput, latency, or loss of user traffic. Instead, OrbWeaver uses the selective amplification method described below.

① Packet generation. The IDLE stream generation of OrbWeaver begins with a low-rate but predictable stream of generated IDLE packets. The focus of this process is to provide a ‘seed’ stream with an emphasis on regularity; amplification up to T occurs later in the pipeline. More specifically, the generator module is configured to send a packet every $\frac{\tau_{\min}}{2}$ secs, where τ_{\min} is the minimum τ_i of any port on the pipeline.

There are two important aspects of this seed stream. The first is that the rate is double that of τ_{\min} in order to provide a degree of oversampling for the subsequent optimizations without sacrificing guarantees on the eventual spacing of packets. The second is that the IDLE packets are configured with a strict high priority at the ingress arbiter so that the packet will always be serviced as soon as it is generated. While this implies that IDLE packets are preferred over user traffic in the ingress pipeline, the low rate of this seed stream means that OrbWeaver incurs <1.5% overhead even for the worst case of minimum-sized packets sent at 100 Gbps (denoting the optimal τ_i for a 100 Gbps link). More typical packet sizes and utilization eliminate the overhead.

② Amplifying the stream on-demand. OrbWeaver takes the low-rate seed stream above and amplifies it, potentially up to the full rate T , by leveraging another hardware feature found in modern switches: flexible multicast. In Figure 2, this behavior is implemented in the PRE, which can replicate a packet descriptor to the egress queues at line rate.

Unfortunately, the naive approach of replicating a packet to every egress queue every τ_{\min} seconds can crowd out normal multicast packets and waste significant egress capacity. More specifically, there are two instances where it is not necessary to multicast a packet to a particular port i :

1. If the port is slower than the maximum speed, then sending at τ_{\min} will be too fast by a factor of $\frac{B_{\max}}{B_i}$.

2. If a user packet was already sent to the egress port recently, sending an IDLE packet is unnecessary.

OrbWeaver addresses both cases by oversampling the sending history of each port (at rate $\frac{\tau_{\min}}{2}$) and then selectively filtering/multicasting toward only the ports that need an IDLE packet. When a port i has bandwidth $B_i < B_{\max}$, the switch downsamples the IDLE packets by configuring two multicast groups (one with port i and one without) and picking the one with i every $\lceil \frac{B_{\max}}{B_i} \rceil$ packets. Similarly, if a port has sent a packet (user or IDLE) in the past $\frac{\tau_{\min}}{2}$ seconds, we can select a multicast group that does not contain the given port.

Concretely, this filtering step uses a single stateful register entry with a bit width equal to the number of ports attached to the pipeline. In essence, the register is a bitvector where each bit represents whether we have sent a packet to the associated port within the last $\frac{\tau_i}{2}$ seconds. For every incoming seed packet, if the associated bit is 1, we omit the port and flip the bit to 0; if the bit is originally 0, include the port in the multicast and flip the bit to 1. Specifically:

```
user packet: filter_reg |= 1 << egress_port
seed packet: filter_reg ^= speed_mask
```

When all ports are the same speed, `speed_mask` is always $2^N - 1$; for hybrid configurations, the i th bit is 1 for every $\lceil \frac{B_{\max}}{B_i} \rceil$ packets and 0 otherwise. After updating the register, OrbWeaver multicasts the current seed packet to the multicast group specified by `filter_reg` (in particular, its value before the xor)—if and only if bit i in the multicast group ID is 0, port i is included in the multicast.

In principle, a direct application of the above filtering step guarantees that the PRE will have enough bandwidth for all user multicasts, assuming that each user multicast results in at most one packet on each egress port. Two aspects of modern switch design potentially complicate this design.

The first is that today’s switches typically cannot support a unique multicast group for each of the 2^N possible combinations of target ports. OrbWeaver addresses this by reducing the number of groups by coalescing ports into groups of M such that, if any port in the set has its bit in `filter_reg` set, the entire set receives the multicasted packet. This approach trades a factor of 2^M reduction in the number of multicast groups for a worst-case $\frac{M-1}{N}$ -factor decrease in PRE bandwidth. The second is that modern switches are often composed of different pipelines, each supporting distinct packet generators, sets of registers, and groups of ports. Lack of visibility across pipelines means that `filter_reg` may only track local sends, which can also lead to higher PRE usage.

We note, however, that in both of the above cases, OrbWeaver will only incur false negatives (and no false positives) of user packet presence, thus satisfying R1. We also note that very few modern networks are continuously multicasting to all ports at near line-rate.

- ③ Weaving the IDLE stream between user packets.** After the stream is amplified, it reaches the egress queues and

pipeline of the switch. To bound the impact of the stream on user traffic, OrbWeaver configures its packets to have a strictly lower priority than any other user traffic on the same port. If there is user traffic to send, the IDLE packets will not impact them; if there is no traffic to send, the IDLE packets will be sent at a minimum rate of τ_i per port i . The only potential impact to the latency/throughput of user traffic is when an IDLE packet is scheduled just before a user packet arrives, in which case the user packet will be delayed by at most $\text{pkt_size}/B_i$. The delay is only incurred once per packet burst, which implies a bound on OrbWeaver’s end-to-end impact on latency and throughput.

Upon arriving at the ingress pipeline of the downstream switch, the packets will be dropped. This also has near-zero impact on user traffic as IDLE packets are only received when the upstream switch has nothing to send.

④ Managing the packet buffer and egress queues. Finally, through the above process, there are two primary places where IDLE packets can compete with user packets for memory in addition to bandwidth. The first is the per-egress output queues that hold packet descriptors before they are serviced by the egress pipeline. The second is the shared packet buffer that stores packet contents until they are sent out on the wire.

To bound the impact of OrbWeaver on both resources, we statically carve the buffer using egress and ingress buffer accounting mechanisms, respectively. For the former, we note that the queue for IDLE packets (the lowest priority queue for the port) is distinct from those of user packets. This queue only needs to be one cell deep as another IDLE packet is guaranteed to arrive in a timely fashion, and thus, the impact on aggregate memory capacity is negligible. For the latter, we can likewise keep the required buffer shallow because of the guarantees of the packet generation process. Specifically, we can confine the IDLE packets to a fixed-size, non-shared region of the packet buffer. The buffer only needs to have a depth equal to the sum of the egress, per-port IDLE-packet queues plus a small amount of headroom for any potential cycle-level processing delays. This is $< 0.01\%$ of the total buffer size of a typical modern switch.

3.2 Evaluating the Weaved Stream

In this section, we delve deeper into OrbWeaver’s potential impacts on user traffic. We do this with the assistance of a prototype implementation on a $2 \times$ Wedge 100BF-32X testbed. Additional experiments can be found in Appendix F.

3.2.1 Can OrbWeaver Inject at Rate T ?

To demonstrate that our approach can achieve T on a fully provisioned switch, we validate it empirically. Specifically, we configure a switch with all 32 ports active and running at a full 100 Gbps. We then configured the switch’s packet generator module to generate seed packets at a rate of $2/\tau_{100\text{Gbps}}$ and then multicast every other IDLE packet to all ports.

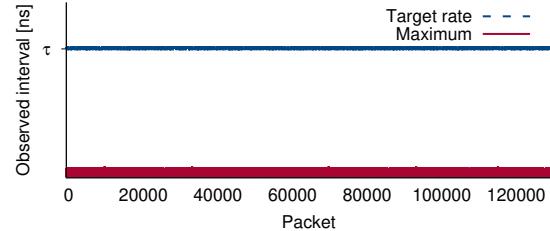


Figure 3: An empirical evaluation of the switch’s capacity to generate IDLE packets. Packets were injected to all ports, but the graph depicts the observed inter-packet gap at only one of those ports. Results are shown for both the target rate ($B_i = 100 \text{ Gbps}$, MTU = 1500 B) and the maximum achievable rate. y-axis omitted to protect confidential information.

Figure 3 shows a time series of the interval between IDLE packets, as observed by the egress pipeline of a single port. To record the series, we maintained a ring buffer (implemented via a data plane register) of the difference between the current `egress_global_tstamp` and the previous. The observations were maintained in the egress pipeline and for a single port (other ports’ results are identical).

We find that, not only is the injected stream able to achieve $\tau_{100\text{Gbps}}$ for every port simultaneously, the observed rate is stable across packets. Further, increasing the amplification factor of the multicast configuration enables IDLE packet generation more than an order of magnitude faster than the target interval, $\tau_{100\text{Gbps}}$. Among other implications, this means IDLE packet injection is robust to higher bandwidth and lower MTUs, even without improvements to packet replication capacity.

3.2.2 Can OrbWeaver Bound Packet Gaps?

In addition to being able to generate IDLE packets at rate T , R1 also requires regularity in the form of a bound on the gap between packets. We note that Figure 3 already demonstrates the regularity of this gap on a switch without traffic. We also note that in the other extreme (when ports are always congested), R1 is trivially satisfied.

In this section, we extend these results to a network with burstiness and varying levels of traffic. Specifically, we use a hardware testbed consisting of two OrbWeaver-enabled switches (A and B) and a set of servers connected to A . User traffic is passed hosts $\rightarrow A \rightarrow B$ with amplification to fully utilize the ports at B . For this experiment, we used `tcp replay` and `pcap` traces from an ISP backbone [9] and a cloud data center [8]. We set up a register in the ingress pipeline of the downstream switch B to record the distribution of the interval between consecutive packets.

Figure 4 shows the results for a single 25/100 Gbps port. Without OrbWeaver, very few intervals are under τ for the target link speed, and the tail is very long. OrbWeaver, on the other hand, is able to weave in IDLE packets to guarantee an upper bound on the packet interval regardless of the original traffic pattern. In particular, for a configured generation interval of t ns, out of 2.14×10^9 interarrival periods, the max-

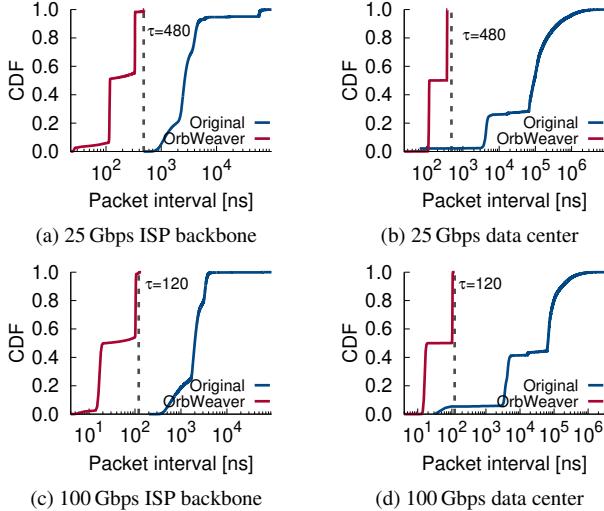


Figure 4: Observed intervals between packets with/without OrbWeaver’s weaved stream. The dotted line shows the ideal period τ for each link speed. Without OrbWeaver, the maximum interval was >100s of ms but we truncate for readability.

imum observed interval was $(t + 3)$ ns (observed for only 32 intervals). The discrepancy is likely due to either clock drift or the aforementioned cycle-level processing delays. Notably, the presence or absence of cross traffic had negligible effect on the frequency of these 3 ns outliers so in practice, we can set $t = \tau - 3$ and achieve reliable results.

Explanation. The regularity of OrbWeaver’s weaved stream derives from the architecture of the switch and the mechanisms of OrbWeaver. From the components of Figure 2, the parser used by the packet generator is separate from those of the external traffic, the ingress pipeline grants strictly higher priority to the generated packets over external traffic (user or IDLE), and the packet buffer protects IDLE packets from interference through static reservations for worst-case capacity. When combined, a generated IDLE packet can only be delayed through HoL blocking when an external packet arrives just before the generated packet. For unicast packets, this is a 1-cycle delay; for full broadcasts, this is up to an N -cycle delay (which is short for today’s high-speed networks).

At the egress pipeline, the priorities are reversed: IDLE packets are set to a strictly *lower* priority than user traffic. This change stems from a change in objective: in the egress pipeline, it is no longer necessary for the IDLE packets to be sent at a precise rate; instead, the goal is to send *any* packet at above the minimum rate, τ_i . Choosing a user packet instead of an IDLE one can only decrease the inter-packet gap.

Note that, in a Tofino, these priorities (unlike those at the ingress) are only effective within their respective ports. Thus, the switch will send a low-priority packet on port i even if there is a higher-priority packet queued for a different port. As long as the average packet size is above the minimum for line-rate processing, ports can be considered in isolation.

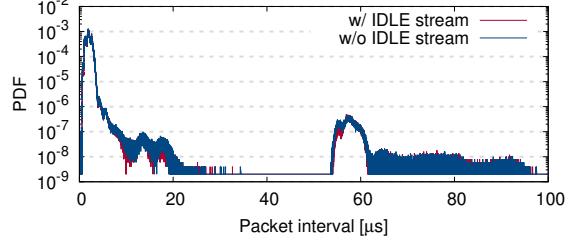


Figure 5: The impact of IDLE packets on user traffic at the ingress pipeline with/without a generation rate of $2/\tau_{100}$ Gbps.

3.2.3 Do IDLE Packets Affect External Traffic?

As important as the impact of cross traffic on generated IDLE packets are the impacts of the generated packets on (1) user traffic and (2) incoming IDLE packets. A significant impact on (1) implies violations of R2; on (2), it implies inaccuracy in inter-arrival times and potential violations of R1. We discuss potential impacts in the two pipelines separately.

Ingress pipeline. While OrbWeaver’s packet prioritization means that IDLE packets will be preferred over external traffic in the ingress pipeline, its use of multicast amplification reduces their impact to 1.5% of maximum packet-level capacity, with zero impact to byte-level capacity.

To evaluate the practical effects of this overhead, we replayed a real-world packet trace over an ingress pipeline of an OrbWeaver switch. The packet trace was generated using `tcpreplay` and link-level packet traces captured from 10 Gbps Internet routers [9]. To saturate the pipeline, we sped the traces up to match our setup’s 100 Gbps per-link bandwidth and replicated them to fill the switch.

We compare two cases. In the first, only the above external traffic is present. In the second, we used the exact same traces but, in parallel, we injected IDLE packets into the same pipeline just as we did in the previous subsection. In both cases, we measured the packet count and interarrival times of user packets in the ingress pipeline with the help of stateful registers that aggregate both statistics.

We find that, for the speeds and packet sizes in the evaluated trace, the throughput and congestion loss of user traffic is the same whether the generated IDLE stream is present or not. The only metric that is impacted is latency, where a slight delay can be introduced each time a generated packet is processed one ‘clock cycle’ ahead of a user packet; however, this is minor and mitigated by the low frequency of IDLE packet injection. Figure 5 depicts the cumulative impact of this delay using a histogram of the packet interarrival time of the traces, with and without the IDLE stream—the majority of the differences are due to randomness in `tcpreplay` between executions, rather than OrbWeaver.

Egress pipeline. The benefits of the amplification strategy to contention mitigation stop at the PRE, but two other factors take its place in ensuring that user traffic is not impacted in the egress pipeline. The first factor is the filtering step that was introduced in Section 3.1, which prevents superfluous

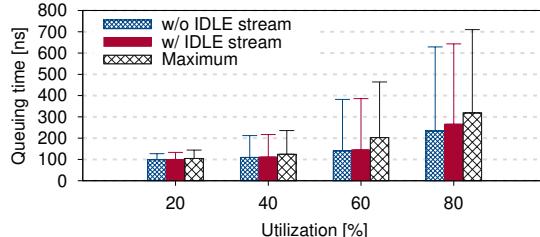


Figure 6: The impact of IDLE packets on the latency of user traffic at the egress pipeline. Results are shown for various levels of average utilization. 0% and 100% are not shown as OrbWeaver becomes trivially optimal. To provide an upper bound on the impact of the IDLE packets, we disable adaptive ingress filtering and populate the pipeline with only small (64 B) user packets. A real OrbWeaver deployment would have much lower impact.

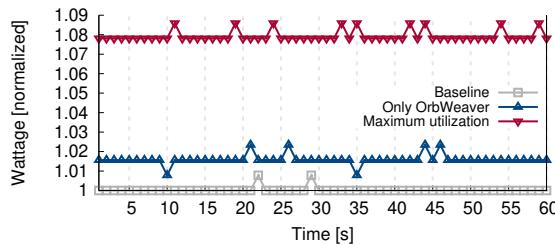


Figure 7: The power draw of a OrbWeaver switch compared to that of an idle (baseline) and a maximally utilized switch. Y-axis is normalized to the average power draw of the baseline.

usage of both the PRE and egress pipeline when the egress ports are already occupied. For IDLE packets that are not filtered in the ingress pipeline, the second factor is the strict prioritization of user traffic over IDLE packets of the same port, also introduced in Section 3.1. The second factor, in particular, provides an upper bound on the impact of the IDLE packets as long as the user traffic respects the minimum average frame size requirements of the switch specification (see Appendix D for a formal analysis).

To truly stress these mechanisms, we evaluate an extreme scenario in which multiple hosts send minimum-size (64 B) packets toward a single egress port and OrbWeaver’s filtering mechanism is disabled. This situation is not possible in OrbWeaver, but is helpful in demonstrating the efficacy of egress prioritization for protecting user traffic. The results verify the analysis above, even for high user-traffic utilization. For comparison, we also show the impact of an IDLE stream operating at the order-of-magnitude-higher maximum rate of Figure 3 but still set to low-priority. Again, across all experiments, throughput was unaffected.

3.2.4 Does Injection Affect Power Usage?

Finally, we investigate the impact of weaving on the power consumption of today’s switches. A natural concern is that the continuous stream of packets will increase consumption; however, we find the actual impact is minimal as the underlying Ethernet MAC already continuously sends IDLE symbols.

To evaluate this, we used a P3 Kill-A-Watt Electricity Us-

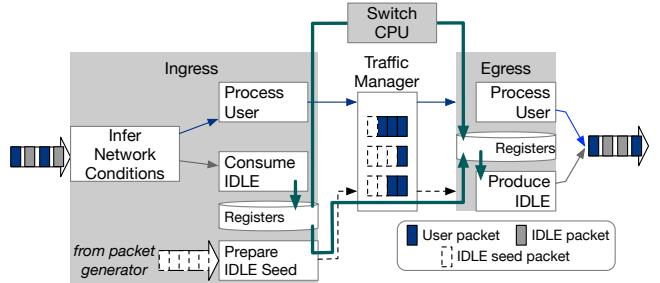


Figure 8: Structure of a P4 program that processes a weaved stream. The ingress pipeline extracts information from the weaved stream, then processes user and IDLE packets separately. The egress pipeline processes user packets and transforms seed packets into IDLE packets. Pipelines can communicate using registers that are synchronized with either seed packets or the switch CPU, as shown by the thick lines.

age Monitor (Model P4400) to measure the total power draw of a Wedge100BF-32X programmable switch. The monitor sits between the switch’s power plug and its power outlet and can measure wattage to within 0.2–2.0%. To emulate the switch’s deployment into a network of programmable switches, we connect every port on the switch to a second switch that logically functions as 32 neighboring switches. We test three distinct configurations:

- *Baseline*: All ports on the switch are connected at 100 Gbps; however, the switch is otherwise inactive, i.e., there is no incoming traffic nor any IDLE packets.
- *Only OrbWeaver*: Same as above, but with OrbWeaver’s IDLE stream generation enabled on all switches. The switch is, thus, both sending and receiving packets at T .
- *Maximum utilization*: The worst case scenario, where the switch is both sending and receiving user packets at the maximum rate and generating IDLE packets (that are eventually dropped in the ingress pipeline).

Figure 7 shows the power draw of each configuration over a 1 min period. OrbWeaver’s transmission of packets at rate T increases the average power draw of the switch by <2%.

4 Use Cases

Figure 8 outlines the general structure of a P4 program that uses OrbWeaver. Whereas a standard P4 program processes a stream of user packets, an OrbWeaver P4 program processes a weaved stream of user and IDLE packets. OrbWeaver programs can append/read information from the payloads of the IDLE packets (which appear on the wire as a special Ether-Type) or infer statistics from the timing of the weaved stream. In either case, the content of IDLE packets can be manipulated just like any other packet (metadata like the drop decision, priority, or egress port should not be changed).

In typical usage, the receiving switch will process, record, and drop incoming IDLE packets before the end of the ingress

pipeline. In most cases, the IDLE packets bypass the normal pipeline logic and, thus, will not affect user byte/drop/error counters. Separately, they use either (a) an agent on the switch CPU [47] or (b) a locally generated IDLE seed packet to transfer data from the ingress to the egress pipeline before sending to the downstream switch. Together, they facilitate multi-hop communication over IDLE packets.

In this section, we detail three example use cases of OrbWeaver (see Appendix A for others). For each example, we consider a recently proposed network system, and we explore how well OrbWeaver can approximate it without introducing any additional impact on user traffic. We note that, in some cases, this restriction can result in suboptimal designs (i.e., imposing on user traffic may result in better overall performance, even if it incurs overhead). Rather, we ask: how far can operators go before needing to ever consider the choice between network throughput and features?

4.1 Use Case #1: Fast Failure Detection

Failures of network components are common in large networks where the number of devices involved ensures a constant flow of incidents. Reasons for the failures include overheating components, power instability, bit flips in the signal, loose transceivers, bent fibers, or any number of other causes [15, 44, 51, 52]. In the end, however, the symptom of many of these failures is the same: lost packets in the network.

Thus, as the first steps toward mitigation, quickly detecting and quantifying packet loss is critical to maintaining high availability and stringent SLOs, particularly as networks improve in both bandwidth and reaction time such that control-plane processing is no longer the sole bottleneck [11, 26, 30–32, 47]. Unfortunately, as mentioned in Section 2, common detection approaches—periodic keepalives or pings—force network architects to sacrifice detection latency to constrain overheads. Moreso as pings are traditionally prioritized over user packets to minimize false positives.

Even recent systems like NetSeer [50] that track user-packet loss inband (without injecting additional packets) suffer from this tradeoff. For example, NetSeer’s choice to not inject additional packets means that the network is necessarily slow to detect a black hole (differentiating from a lack of demand requires CPU coordination to compare the flow counters of adjacent switches). Likewise, their choice to tag every packet with a sequence number incurs a bandwidth overhead of 0.3%~6.3% in return for higher detection granularity (unless there are previously unused bits in the header and we cannot change the data plane to remove them).

4.1.1 An OrbWeaver Redesign

Taking NetSeer as a base, we can replace its inter-switch communication with an OrbWeaver-influenced design to eliminate bandwidth overheads and significantly improve detection time. We refer readers to the original paper [50] for full details of the existing system but summarize the relevant components

as follows. NetSeer records the 5-tuple of each packet in the egress pipeline using per-port ring buffers and tags it with a 4-byte sequence number. The downstream switch stores the last observed sequence number. Upon detecting a gap (e.g., packet 14 after packet 12), it sends 3 duplicate and high-priority drop notifications to the upstream switch for each missing sequence number. If the upstream switch receives at least one such notification, it will use the records in the ring buffer to generate a flow event for the missing packet, which will be compressed/summarized for the management plane.

In NetSeer-OW, switches maintain per-port hash tables that, like NetSeer, record the 5-tuples and packet counts of passing flows (using the 5-tuple hash as the index). The caches are maintained in the egress pipeline of each upstream switch as well as the ingress pipeline of each downstream switch. As channels are FIFO and the tables use the same size and deterministic hash function, their content should always be identical. The only exceptions occur after a packet loss, at which point either a counter or a 5-tuple will differ.

In this re-design, user packets are *not* tagged with any additional data nor does it require triple-notifications. Instead, the upstream switch will opportunistically embed in IDLE packets pseudo-randomly selected cache records². If the downstream switch finds that a record differs from its local copy, it will generate an event for the contained 5-tuple. It will also generate an event if packets stop arriving, which is detected with locally generated IDLE seed packets that scan per-port weaved-stream counters. After NetSeer-OW compresses/filters these events, the control plane sends the results over a low-priority TCP connection to the central controller.

Note that, in addition to exploiting the IDLE stream to carry flow information, (R1)’s guarantee of packet arrival rates enables provably optimal detection speed of link failures. In principle, OrbWeaver can trigger an alert if the `ingress_mac_tstamp` of any two consecutive packets is $\leq \tau$. While that level of granularity may be too aggressive for many networks, we note that recent proposals for data plane rerouting have made detection speed a bottleneck [11, 26, 30, 32], particularly if a goal is zero-loss failure recovery. In the end, the point is that OrbWeaver can provide arbitrarily precise failure detection/statistics for current and future networks.

Dealing with a lack of sending opportunities. While extended periods of maximum utilization are rare in most networks [9, 38, 49], NetSeer-OW can still provide useful properties during these extreme conditions. For example, for failure detection, a downstream switch in a fully utilized network can immediately detect a packet drop by examining the gaps between adjacent packets (a drop occurred when the gap $> \tau$).

Flow attribution is slightly more challenging, with the chief concern that the switch evicts the flow before including it in an IDLE packet. We can quantify the probability of this hap-

²To improve the update rate, we can pack up to three 5-tuple-counter records (IPv4 and counters of 3 B) in each packet. To handle register access limitations, we can pack the records or split the table across multiple arrays.

Data structure size (per-port)	NetSeer		NetSeer-OW	
	256	64	512	128
SRAM (KB)	384	192	896	320
Number of sALU/register arrays	6	6	7	7

Table 1: Data plane resource usage for typical NetSeer and NetSeer-OW configurations on a 64×100 Gbps switch.

pening using the formalization in Appendix E. For reference, using the assumptions of Appendix E, average utilization of [9, 38], and flow cache performance of [39], ISP routers with 128 cache entries per port would have a $P(\text{notified}) \approx \frac{0.72}{0.72+0.28*0.45/3} = 94.4\%$. A data center switch with 128 cache entries would have $P(\text{notified}) \approx \frac{0.75}{0.75+0.25*0.16/3} = 98.2\%$, or with 512 entries $P(\text{notified}) \approx \frac{0.75}{0.75+0.25*0.05/3} = 99.4\%$.

Benefits. Compared to the original NetSeer design, the primary benefit of the OrbWeaver augmentation is to completely eliminate *all* sources of bandwidth overhead—in essence, we can apply NetSeer for ‘free.’ In particular, it eliminates the overhead of sequence number tagging (0.3%~6.3%) of capacity; the replicated, high-priority failure notifications (up to 100% of reverse link capacity); and the impact on user traffic of the event reports. Beyond overhead, it also improves the speed to detect inter-switch failures, particularly during periods of low utilization.

Table 1 shows the data plane memory consumption of both systems. Additional memory increases $P(\text{notified})$, however the relationship is different for each system. As a concrete data point, consider the coverage goal highlighted in the original NetSeer evaluation [50]—to correlate 90% of packet loss events with flows. For a 64×100 Gbps switch and a similar estimation strategy as above, NetSeer-OW meets this goal with 320 KB of SRAM (128 cache slots per port) in both ISP and data center workloads. On the other hand, assuming the network’s minimum packet size is 64 B, NetSeer requires approximately 384 KB of SRAM to meet the 90% coverage objective because it must allocate enough ring buffer slots per port (256) to ensure that sequence numbers are not overwritten before switches have a chance to correlate their results.

4.1.2 Evaluation

Detecting failures more quickly. To quantify how quickly NetSeer-OW can detect a failure, we deployed NetSeer-OW to a hardware testbed and randomly disconnected a link between the two switches A and B 100 times to emulate 100 fail-stop link failure events. To test the limits of our approach, we configured the probes to mark a τ -timeout failure as soon as even a single packet loss is detected.

Figure 9a shows the detection time of trials for 10, 25, and 100 Gbps links. NetSeer-OW achieved 100% precision and recall. It also consistently detected the failure within 10s of nanoseconds of the optimal time. In contrast, typical configurations for protocols like Bidirectional Forwarding

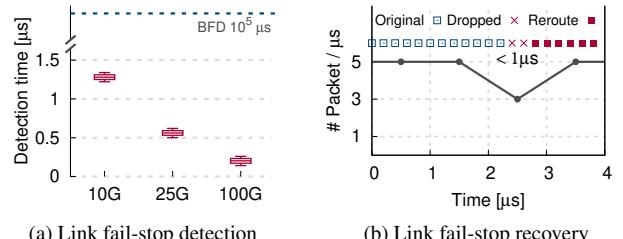


Figure 9: (a) the min, Q_1 (p25), median, Q_3 (p75), and max of OrbWeaver’s time to detection across 100 failure events. (b) OrbWeaver’s time to recovery (<1 μ s) from a bidirectional failure of a 25 Gbps link. A total of two packets are lost.

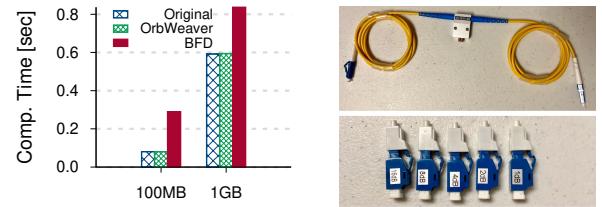


Figure 10: (a) shows the transfer completion time comparison for original, NetSeer-OW, and BFD (100 ms) in a simple leaf spine topology. With NetSeer-OW’s fast detection and data plane reroutes, the impact is minimal.

Detection (BFD) are closer to 10s or 100s of milliseconds; even recent data plane detection systems [20, 30] are several orders of magnitude slower than NetSeer-OW can achieve.

Figure 9b shows the resulting seamless recovery when NetSeer-OW is combined with a simple data plane rerouting mechanism. In the experiment, we induce a bidirectional failure in one link between A and B, and we configure B to failover to a backup path as soon as it detects an error. On top of this setup, we send a steady stream of packets on the target link at a relatively high rate of 5M packets per second. A total of two packets were lost—likely in-flight.

End-to-end impact. To evaluate the end-to-end impact, we emulate a leaf-spine topology with 2 leaf switches L1, L2 and 2 spine switches S1, S2. All switches run OrbWeaver with pre-computed data plane backup paths. Between L1 and L2, we insert a variable fiber optic in-line attenuator capable of 0~60 dB attenuation. On hosts connected to the leaf switches, we run TCP transfers of varying sizes using iperf, during which we increase attenuation from zero until failure and examine the impact over the transfers experiencing the events. As Figure 10a shows, with OrbWeaver, the impact of failure is negligible with respect to completion time. In contrast, with BFD, failures cause the 100MB transfers to take over 4× longer and the 1GB transfers to take over 30% longer.

4.2 Use Case #2: Time Synchronization

Time synchronization is another common task in modern networks. Like failure detection, time synchronization requires

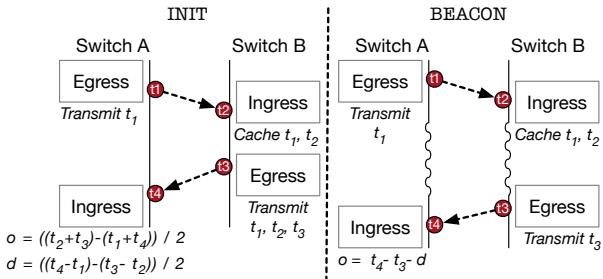


Figure 11: Time sync in DPTP-OW, using IDLE packets. When the difference between t_2 and t_3 is small, A treats the message as part of an INIT phase and calculates o , the clock offset, and d , the one way delay. When it is high, the BEACON phase uses the most recent d to track clock frequency drift.

coordination between adjacent switches, and many other applications rely on its accuracy [13, 36, 45, 46].

Unfortunately, the most common methods for synchronizing time between adjacent machines involve the computation of One-Way Delay (OWD) using periodic, high-priority echo requests/replies [1, 14, 30]. Here too, architects are presented with a tradeoff: clock frequency drifts imply that the faster we send echoes, the more closely we can bound the clock offset and the more accurate the synchronization. Protocols like DTP [29] that integrate the protocol into the physical layer can circumvent this overhead but require hardware changes.

4.2.1 An OrbWeaver Redesign

The state-of-the-art in time synchronization for programmable switches is DPTP [25]. In it, two adjacent switches (a client, A, and a server, B) compute the offset of their local clocks by leveraging switches’ ability to embed timestamps into each packet during different stages of packet processing. Host and multi-hop synchronization are also possible using multiple strata. The protocol calls for three messages in each round of the protocol: (1) a DPTP request [$A \rightarrow B$], (2) a DPTP response [$B \rightarrow A$], and (3) a DPTP follow-up [$B \rightarrow A$]. All three messages are high-priority to eliminate queuing delay.

(1) is timestamped using the Tofino egress_deparser_tstamp and ingress_mac_tstamp of A (t_1) and B (t_2), respectively. (2) is timestamped using the same counters in B (t_3) and A (t_4), respectively. In a traditional clock synchronization protocol, the offset would be computed as $\frac{(t_2 + t_3) - (t_1 + t_4)}{2}$. Unfortunately, we note a fundamental limitation of today’s programmable switches—that the egress_deparser_tstamp does not capture the actual point of packet serialization. Thus, the computed offset is subject to variable delays as a result of egress MAC contention. As a result, DPTP introduces the third packet, the follow-up, which embeds a more accurate egress serialization timestamp (obtained out-of-band). Again, we refer interested readers to [25] for full details.

An OrbWeaver-inspired redesign can obviate the need for the third, follow-up message by inferring the egress MAC contention from the weaved stream (and only using results

with no contention). This allows us to use the traditional two-way protocol of Figure 11. It can also eliminate the impact of the remaining messages using opportunistic sends.

Opportunistic synchronization: Rather than relying on high-priority echoes, a system can rely solely on OrbWeaver’s IDLE packets to piggyback timestamps. In particular, whenever A has an opportunity, it sends a request to B on an IDLE packet with a field for t_1 . Upon receiving the packet, B maintains a cache for the most recent values of t_1 and t_2 . Separately, whenever B has an opportunity, it sends the most recent values of t_1 and t_2 along with the local egress_deparser_tstamp in t_3 . In an empty network, A can calculate the clock skew as $\frac{(t_2 + t_3) - (t_1 + t_4)}{2}$ just as DPTP but with much more frequent synchronization (leading to lower jitter, i.e., nominal error [33]).

A challenge with the above approach occurs in networks with high utilization. The traditional OWD estimation method used above implicitly assumes that the clock drift is constant for the duration of the protocol round; otherwise, the delays at the time of the request and response may not be comparable due to clock frequency drift. In OrbWeaver, this can happen if there is congestion from B to A; the gap between t_2 and t_3 can be unbounded, leading to inaccurate results.

We address this challenge by borrowing an idea from a different protocol, DTP [29]: the decoupling of synchronization into INIT and BEACON rounds. If the time between t_2 and t_3 is sufficiently small, the round is treated as an INIT round and A computes the offset as above. Otherwise, A treats the message as part of a BEACON round where it takes d , the OWD computed from the last INIT round ($d = \frac{(t_4 - t_1) - (t_3 - t_2)}{2}$), and it computes a new offset: $o' = t'_4 - t'_3 - d$.

Selective synchronization: Finally, to remove the need for DPTP’s third ‘follow-up’ message, we can exploit the implicit information contained in the woven stream’s timing. The underlying intuition is simple: if the gap between an IDLE packet and its preceding packet is less than τ , the IDLE packet may have encountered contention at the egress MAC. In this case, the packet’s timestamp may be unreliable. DPTP corrects for this contention with the follow-up message; OrbWeaver simply ignores these protocol rounds. While this filtering effectively requires that usable gaps be $> \tau \sim 2\tau$, it greatly improves the accuracy of the protocol while still permitting frequent re-synchronization in modern networks.

Dealing with a lack of opportunity to send. The above protocol fully synchronizes switches when both links have concurrent IDLE gaps. The protocol also includes support for correcting small drifts when only one direction has a gap (by adjusting to the fastest clock in the network). We note that in a network with multiple paths, we can configure synchronization to propagate among any one of those paths. Thus, if we view the network as a directed graph, the only time a switch may lose synchronization is if sufficient links are maintaining 100% utilization that the links form a cut of the graph. In the end, if operators need assurances, they may need to send

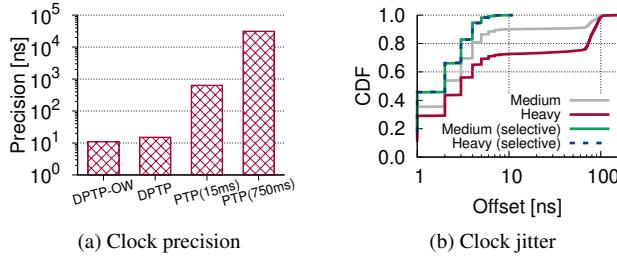


Figure 12: (a) shows the precision for different synchronization protocols and a heavy workload ($\sim 80\%$ CAIDA user traffic). (b) shows the CDF of observed offsets (absolute value) for DPTP-OW upon medium and heavy loads for 10 Gbps link ($\tau = 1200$ ns), w/ or w/o selective sync. OrbWeaver achieves a precision of 11 ns even under heavy user traffic.

higher-priority messages if too much time elapses; however, we can extend our techniques so that the messages only need to be prioritized above the lowest-priority user traffic—high-priority, interactive applications would be unaffected.

Benefits. As long as there is occasional usable bandwidth in the network, OrbWeaver again eliminates all bandwidth overheads without sacrificing accuracy or nominal error. When the network is underutilized, it actually provides similar re-sync intervals as DTP but using commodity PISA switches.

4.2.2 Evaluation

Following prior work, we evaluate DPTP-OW’s precision [29, 43] (defined as the maximum clock skew in the network), as well as its jitter [33] (defined as the distribution of measured offsets or nominal error). Again to match prior work, we evaluate these in a two-switch testbed during a 20 min collection for 10 Gbps link with a medium workload (a CAIDA trace with 25% average utilization) and a heavy workload (the same trace sped up to $\sim 80\%$ average utilization). We compare to both DPTP (with 2000 requests/sec) and PTP. For PTP, prior work has suggested message frequencies ranging from 15 ms to 2 s [1, 2, 29, 30]; we pick two points in this range: 15 ms as a lower bound and 750 ms per the evaluation baseline [29].

We observe that, even at high loads, DPTP-OW can achieve 10 ns bounds in both precision (Figure 12a) and jitter (Figure 12b) without imposing on user traffic. These bounds are similar to or better than DPTP, which incurs high-priority bandwidth overhead. Preliminary tests on higher-link speeds indicate that precision will only improve as τ decreases. In Figure 12b, we further observe that selective synchronization is an effective technique to reduce the message complexity of the protocol while maintaining low jitter and good precision.

4.3 Use Case #3: Congestion Feedback

Finally, many modern networks rely on robust load balancing algorithms to efficiently utilize their multiple paths. There are numerous approaches to load balancing, but among them,

adaptive approaches [3, 26] are attractive as they can react to current network conditions when making balancing decisions.

A state-of-the-art approach is taken by HULA [26], which proposes a protocol for adaptive data center load balancing using programmable switches. In HULA, every switch maintains two tables: a `bestHop` table that stores the best next-hop to each destination ToR, and a `pathUtil` table that stores the utilizations of those next-hops. Destination ToRs periodically flood the network with high-priority probes that traverse all paths (in the reverse direction, dst-to-src) and track the bottleneck link utilization of the best such path—intermediate switches update their `bestHop`/`pathUtil` tables accordingly.

As in the previous use cases, congestion feedback mechanisms like the one in HULA force a tradeoff between overhead and the availability/freshness of congestion data. HULA eventually sets the probing interval to 1-RTT and makes a case for why that is a good tradeoff, but OrbWeaver can potentially provide similar performance using only opportunistic sends.

4.3.1 An OrbWeaver Redesign

An OrbWeaver-inspired redesign replaces the high-priority HULA probes with OrbWeaver’s opportunistic IDLE packets. There are two new challenges. The first is building a ‘flood’ communication model on top of OrbWeaver’s opportunistic sends. The second is dealing with congestion on the reverse path and the resulting lack of new information.

Per-path propagation: For any path through the network, there are two types of hops: ingress-to-egress hops (that bridge the pipelines of a local switch) and egress-to-ingress hops (that bridge adjacent switches).

For the former, HULA-OW leverages the switching ASIC’s PCIe interface to asynchronously mirror the `pathUtil` table between the ingress and egress pipelines of a single switch. We use Mantis [47] to mirror the registers, which completes a mirror operation every ~ 20 μ s without impacting data plane throughput. For the latter type of hop, the system simply sends the contents of `pathUtil` using IDLE packets. To make this process more efficient, we can stripe the `pathUtil` table across m registers and pack m (`dstToR`, `pathUtil`) records into each IDLE packet round-robin style. In an unloaded network, the full table is transmitted in $\frac{R\tau}{m}$ time, where R is the number of ToRs in the data center. We note that even for $R = 1000$ and $m = 1$ (i.e., an unoptimized update rate), this is still more frequent than HULA.

Stale information: If there is persistent congestion on the reverse path, utilization information may not be able to propagate across the network; the switch adjacent to the congestion will know the utilization of the adjacent link, but not downstream links. To handle this case, HULA-OW uses a simple aging mechanism. Specifically, it will track the EWMA of all observed `pathUtil` values for every destination ToR (in addition to the minimum). After each RTT with no information from the best path, it will shift the best path’s `pathUtil` value toward the average (with a lower bound of the adjacent link’s

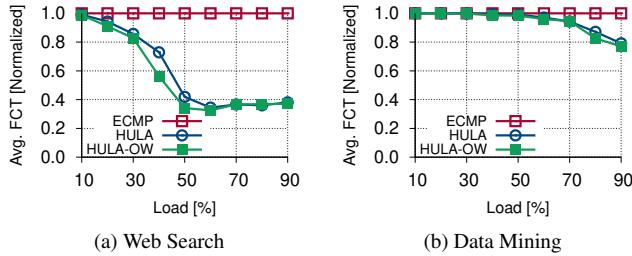


Figure 13: Avg. FCT (normalized to ECMP) for HULA and HULA-OW upon different loads of DCTCP and VL2 traces.

utilization). If no information comes from *any* neighbor for several RTT and the adjacent links are all equal, the switch will fall back to random flowlet placement.

Dealing with a lack of opportunity to send. We note that the effect of the above metric-aging strategy is that `bestHop` will be quickly overwritten by the ‘next-best hop’ whose reverse path has opportunities to send. Assuming that at least some congestion information gets through, HULA-OW will still provide substantial benefits due to properties like the power of two choices [34]. In the worst case, it achieves equivalent performance to flowlet ECMP.

Benefits. Across all regimes, HULA-OW eliminates the probe overhead on network bandwidth. In networks with low utilization or high burstiness, it provides more frequent utilization updates than HULA in addition to increasing the peak usable bandwidth (see below).

4.3.2 Evaluation

Performance. We evaluate HULA-OW in NS-2 using the same FatTree topology as the original paper (Figure 4 of [26]). Also like HULA, we leverage synthetic workloads based on web-search [4] and data-mining [16]) and configure HULA to probe at a 200 μ s interval. Figure 13 shows the avg. FCT (normalized to ECMP) for HULA and HULA-OW.

Despite the frequent periods of full utilization in these workloads (especially at high average load), we observe that HULA-OW is able to find sufficient gaps between packets to efficiently transfer utilization information. Overall, HULA-OW is able to provide comparable or better performance than HULA in all of the tested cases, even in the presence of very high average utilization. The performance is also always either equivalent or better than the ECMP baseline.

Overhead reduction. The bandwidth overhead of HULA probes is given by $\frac{\text{probeSize} \times \text{numToRs} \times 100}{\text{probeFreq} \times \text{linkBandwidth}}$ [26]. With 500 ToRs, $\text{probeFreq}=200 \mu\text{s}$, $\text{probeSize}=64 \text{ B}$, and 100 Gbps links, HULA occupies 1.6% of the network’s bandwidth. In contrast, HULA-OW occupies close to *zero* of the network’s usable bandwidth and only 1.5% of the packet-level capacity of the ingress pipeline (which HULA’s probes also consume).

5 Related Work

Leveraging unused resources. OrbWeaver is not the first system to propose the opportunistic use of leftover resources. Indeed, many applications of priorities are in a similar spirit. Even in contexts outside of computer networking, others have used low-priority background tasks and spot VMs to harvest unused CPU cycles and memory [5].

In networking, close related work includes software WANs like SWAN [21] and B4 [24], which divide traffic into classes that range from interactive to background—interactive traffic is given priority while background traffic soaks up any remaining bandwidth. These systems successfully provide opportunistic bandwidth utilization but focus on end-host data. As explained in Section 2, these approaches can leave parts of the network unutilized due to both application traffic patterns and structural bottlenecks. OrbWeaver is, thus, complementary to these approaches and can be used to reclaim the remaining bandwidth for intra-network coordination.

Prior work has also applied similar techniques to lower layers, for instance, in the case of Ethernet’s IDLE symbols or F10’s rapid heartbeats [32]. F10, in particular, proposed a failure detection mechanism that is close to OrbWeaver’s in which devices continue to send traffic even when idle. In comparison, OrbWeaver’s contribution is make the idea practical on high-speed programmable switches, to closely examine the resulting impacts on switch configurations and user traffic, and to show how to seamlessly integrate the weaved stream into a spectrum of applications beyond the use case of F10.

Applications of OrbWeaver. OrbWeaver also builds explicitly on prior work that improves networks with coordination, signaling, and probes. We refer readers to the relevant parts of Section 4 for a discussion of the systems on which OrbWeaver builds, and to the original papers for a more complete examination of related work for our applications.

In general, however, OrbWeaver improves on much of the prior work by providing comparable or better performance with near-zero overhead. Exceptions include systems like F10 [32] and DTP [29], which use hardware support to eliminate protocol overheads. As mentioned above, OrbWeaver’s contribution is to generalize the concept and demonstrate a practical framework for it on commodity network devices.

6 Conclusion

Must data plane applications always choose between coordination fidelity and bandwidth overhead? This paper demonstrates that, somewhat surprisingly, they do not. To that end, we introduce OrbWeaver, a framework for opportunistic coordination in a manner that does not affect user traffic or switch power consumption. Using three recently proposed systems, we show how to leverage OrbWeaver to eliminate their bandwidth overheads while maintaining their efficacy.

Acknowledgments

We gratefully acknowledge Vladimir Gurevich for his assistance in understanding the Tofino switch architecture. We also thank Vladimir, Gianni Antichi, our shepherd Aurojit Panda, and the anonymous NSDI reviewers for all of their thoughtful comments. This work was funded in part by Google, Facebook, VMWare, and NSF grant CNS-1845749.

References

- [1] Ieee standard 1588-2008. <https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4579757>, 2008.
- [2] Juniper precision time protocol overview. <https://www.juniper.net/documentation/us/en/software/junos/time-mgmt/topics/concept/ptp-overview.html>, 2020.
- [3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.
- [4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.
- [5] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing slos for resource-harvesting vms in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, November 2020.
- [6] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 419–435, Renton, WA, April 2018. USENIX Association.
- [7] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. *SIGCOMM ’20*, page 662–680, New York, NY, USA, 2020. Association for Computing Machinery.
- [8] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.
- [9] Caida. The caida ucsc statistical information for the caida anonymized internet traces. https://www.caida.org/data/passive/passive_trace_statistics.xml, 2019.
- [10] M. Chiesa, R. Sedar, G. Antichi, M. Borokhovich, A. Kamisiński, G. Nikolaidis, and S. Schmid. Fast reroute on programmable switches. *IEEE/ACM Transactions on Networking*, pages 1–14, 2021.
- [11] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. Purr: A primitive for reconfigurable fast reroute: Hope for the best and program for the worst. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 1–14, 2019.
- [12] Intel Corporation. P4-16 intel tofino native architecture – public version. Application Note 631348-0001, Intel Corporation, March 2021.
- [13] Thomas G. Edwards and Warren Belkin. Using sdn to facilitate precisely timed actions on real-time data streams. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN ’14, page 55–60, New York, NY, USA, 2014. Association for Computing Machinery.
- [14] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 81–94, 2018.
- [15] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 350–361, 2011.
- [16] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.
- [17] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.
- [18] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’18, page 357–371, New York, NY, USA, 2018. Association for Computing Machinery.
- [19] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 71–85, 2014.
- [20] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 161–176, 2019.
- [21] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM ’13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery.

- [22] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 701–721, 2020.
- [23] Van Jacobson. Compressing tcp/ip headers for low-speed serial links. Technical report, RFC 1144, February, 1990.
- [24] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hözle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM ’13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.
- [25] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 8–20, 2019.
- [26] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.
- [27] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *Demo paper at SIGCOMM ’15*, 2015.
- [28] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: Enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM ’21, page 223–244, New York, NY, USA, 2021. Association for Computing Machinery.
- [29] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherpoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467, 2016.
- [30] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter H Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. 2020.
- [31] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring connectivity via data plane mechanisms. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 113–126, 2013.
- [32] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, 2013.
- [33] Jim Martin, Jack Burbank, William Kasch, and Professor David L. Mills. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, June 2010.
- [34] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.
- [35] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Praateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.
- [36] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM ’14, page 307–318, New York, NY, USA, 2014. Association for Computing Machinery.
- [37] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network’s (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.
- [38] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hözle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network. *SIGCOMM Comput. Commun. Rev.*, 45(4):183–197, August 2015.
- [39] John Sonchack. *Balancing Performance and Flexibility in Hybrid Network Telemetry Systems*. PhD thesis, University of Pennsylvania, 2020.
- [40] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM ’21, page 731–747, New York, NY, USA, 2021. Association for Computing Machinery.
- [41] Charles E. Spurgeon. *Ethernet: The Definitive Guide*. O’Reilly & Associates, Inc., USA, 2000.
- [42] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 571–592, 2021.
- [43] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.
- [44] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 419–430, 2012.
- [45] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 402–416, 2018.

- [46] Nofel Yaseen, John Sonchack, and Vincent Liu. tpprof: A network traffic pattern profiler. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1015–1030, Santa Clara, CA, February 2020. USENIX Association.
- [47] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 296–309, 2020.
- [48] Lior Zeno, Dan RK Ports, Jacob Nelson, and Mark Silberstein. Swishmem: Distributed shared state abstractions for programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 160–167, 2020.
- [49] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, pages 78–85, 2017.
- [50] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 76–89, 2020.
- [51] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.
- [52] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 362–375, 2017.

Application Class	System	Weaved Inference?	IDLE Messaging?	Description
Traffic Engineering	Flowlet load balancing [3, 26]	✓	✓	Section 4.3.
	Performance-aware routing [22]		✓	Propagate route updates in customizable distance-vector routing algorithms using IDLE packets.
	Micro-burst detection [49]	✓	✓	Detect micro-bursts from weaved stream, provide feedback to upstream switches with IDLE packets.
Fault Tolerance	Fast failure recovery [50]	✓	✓	Detect failures (Section 4.1), alert upstream switches with IDLE packets for fast data-plane mitigation [10].
	Consistent replicas [28, 48]		✓	Synchronize eventually-consistent distributed state, e.g., for distributed firewalls, with IDLE packets.
Monitoring	Packet forensics [19]		✓	Transfer packet postcards in IDLE packets to reduce overhead of packet history tracking.
	Network queries [18, 35]	✓	✓	Support queries over both flow and weaved stream statistics, export query results in IDLE packets.
	Latency localization [17]	✓	✓	Measure latency in network core using weaved stream, disseminate measurements with IDLE packets.
Network Services	Clock synchronization [25]	✓	✓	Section 4.2.
	Header compression [23, 42]		✓	Synchronize state of point-to-point packet header compressors with IDLE packets.
	Event-based network control [40]		✓	Carry network control events in IDLE packets.

Table 2: OrbWeaver use cases. A diverse range of data-plane applications can use OrbWeaver’s weaved stream to learn about conditions in the network and/or communicate via IDLE packets that consume no data-packet bandwidth.

A Applications of OrbWeaver

Table 2 surveys 11 applications that can benefit from an OrbWeaver implementation, belonging to four distinct classes. We describe several implementations in Section 4. All applications can be expressed as OrbWeaver P4 programs with the basic architecture shown in Figure 8.

Across all applications, we find that there are two overarching benefits to an OrbWeaver implementation:

1. OrbWeaver’s weaved stream allows data plane applications to infer information about network conditions, such as the presence of congestion or failures in an upstream path.
2. OrbWeaver’s IDLE packet abstraction lets data plane applications disseminate information without consuming user bandwidth. IDLE packets are useful for data transfer between directly connected switches (e.g., to synchronize the context tables of a switch-to-switch packet-header compressor [42]) or across the wider network (e.g., to disseminate information about network faults [32], congestion [49], or even user query metrics [35]).

We note that our focus of these applications and this paper is in-network communication. However, end hosts may also be able to benefit from OrbWeaver, e.g., by examining the output of the weaved stream coming from host-facing ports of ToR switches. Efficient end-host generation of a weaved stream may also be possible, but we leave a full exploration to future work.

A.1 Balancing Multiple Applications

IDLE packets are generated and weaved entirely by the OrbWeaver framework. Applications only embed information and extract it in the receiver. IDLE packets can carry the information of multiple applications. For example, a time synchronization application that needs 12B to carry 4 timestamps can co-exist with a failure detection protocol that needs 48B. In this paper, we assume minimum-sized packets but, in principle, IDLE packets can be MTU-sized with the only effect being a proportionally increased worst-case packet delay. Of course, there are fundamentally a limited number of bytes in each IDLE packet; OrbWeaver leaves the decision on how to allocate these bytes to network architects and operators.

A.2 Preventing Starvation

The primary goal of the paper is to explore the opportunistic use of IDLE cycles for in-network coordination. Because of our opportunistic approach, there may be cases where IDLE packets get starved by user packets; however, as previously noted, two factors mitigate the issue:

- The lack of IDLE packets itself reveals concrete information of the network condition (per R1 guarantee of the weaved stream predictability).
- Prior works observed that persistent user traffic is rare, instead, IDLE cycles (every 10s or 100s of μ s) are ubiquitous.

A wide range of applications can be implemented with only opportunistic communication. Of course, some applications may need additional guarantees, e.g., applications requiring a strict, real-time guarantee w.r.t. minimum rate (i.e., maximum inter-IDLE-packet gap); or applications that need more aggregate bandwidth than the weaved stream can guarantee in a timely fashion.

In these cases, networks can apply a priority escalation mechanism by adding a single register of N (number of ports) slots and check the elapsed time since last seen IDLE packet. Applications can seamlessly escalate the priority of IDLE packets when too much time passes (per the applications' guaranteed rate SLO). In these situations, OrbWeaver still eliminates nearly all overhead in the presence of (micro)bursts, but may impose a fixed overhead during extended periods of congestion.

B Generalization to Other Platforms

Our focus in this paper was on the Tofino family of programmable switches. While a detailed taxonomy and analysis of every programmable platform is out of the scope of this paper, there is reason to believe that other programmable platforms have similar features or can emulate the features needed to implement OrbWeaver.

In particular, OrbWeaver leverages three hardware features of Tofino switches: (1) packet generation, (2) multicast, and (3) packet prioritization. Among these, support for the latter two can be found in almost every modern forwarding device that is designed to handle the Ethernet protocol. Support for onboard packet generation is not as universal; however, one potential solution is to connect a port on each switch to a simple device/CPU responsible for generating regular, periodic packets. Of course, a CPU, even with real-time scheduling optimizations, may not be as dependable as the Tofino packet generator. This may necessitate additional tolerances.

Finally, our conversations with switch vendors indicate that OrbWeaver's mechanisms will scale to future switches with both increased bandwidth and port counts. Part of this is due to the fact that most of OrbWeaver's components scale with the clock rate of the switch and/or are independent to each pipeline. The notable exception is packet generation; however, we note that OrbWeaver currently has more than an order of magnitude of headroom (Section 3.2.1). If MTU transmission time does eventually outpace packet generation latency, OrbWeaver's properties will degrade gracefully.

C Energy-Efficient Ethernet (EEE)

The Ethernet standard contains an optional EEE mechanism [41], which allows switches to transition links into a Low-Power Idle (LPI) mode when there is no data to send.

OrbWeaver may be able provide compatibility by turning off the IDLE stream on a per-port, per-direction basis if there is no user traffic during the past S seconds. Each packet flowing between two OrbWeaver switches would then need a single bit reserved as an 'LPI' indicator. Upon receiving an IDLE packet with the 'LPI' indicator set, a receiver will change its expectation from requiring a packet every τ_i seconds to requiring one every τ'_i seconds ($\tau'_i \gg \tau_i$). The very first user packet after the low-power idle mode will be sent with the 'LPI' indicator unset. Loss can be addressed by again emulating EEE and sending several indicator packets in a row.

Enabling this feature may impact the responsiveness of OrbWeaver applications, but we note that all of the use cases studied can make do with less frequent but still regular coordination. OrbWeaver may be able to synchronize these low-power updates with existing synchronization-maintenance events in the PHY.

D Proof of Priority-effect on User Traffic

Theorem. For an arbitrary user packet size distribution and arrival process, with strict priority scheduling and a measurement time window $T \gg \Delta t$ (Δt denotes transmission time of a single IDLE packet), the throughput of the user traffic is unaffected by the IDLE stream.

Proof. Consider a packet sequence p_1, \dots, p_n with size $\Delta t_1, \dots, \Delta t_n$ and original schedule t_1, \dots, t_n , denote the new schedule upon the coexistence of IDLE stream as t'_1, \dots, t'_n .

We first prove $\forall i \in [1, n-1], t'_i \leq (t_i + \Delta t) \rightarrow t'_{i+1} \leq (t_{i+1} + \Delta t)$. The case for preemptive scheduling is trivially true. We focus on the case of non-preemptive scheduling.

Base case with p_1 : the worst case delay of the transmission is when right at t_1 , an IDLE packet is scheduled to transmit and with strict priority p_1 is scheduled right next to it. Hence $t'_1 \leq (t_1 + \Delta t)$.

For the inductive step, given the new schedule of p_i satisfying $t'_i \leq (t_i + \Delta t)$, we need to show that $t'_{i+1} \leq (t_{i+1} + \Delta t)$. There are three cases for the next packet p_{i+1} :

- $t_{i+1} > (t_i + \Delta t_i + \Delta t)$: at t_{i+1} , the previous packet has finished transmission in the new schedule since $t_{i+1} > (t_i + \Delta t_i + \Delta t) \geq t'_i + \Delta t_i$. The worst case delay is when IDLE packet is scheduled right at t_{i+1} and the transmission is delayed by Δt , i.e., $t'_{i+1} \leq (t_{i+1} + \Delta t)$ holds.
- $t'_i + \Delta t_i \leq t_{i+1} \leq (t_i + \Delta t_i + \Delta t)$: at t_{i+1} , p_i finishes transmitting in the new schedule, similar to the previous case, the worst case is Δt when right at t_{i+1} , IDLE packet gets scheduled, hence $t'_{i+1} \leq (t_{i+1} + \Delta t)$ holds.
- $t_i + \Delta t_i \leq t_{i+1} < t'_i + \Delta t_i$: p_{i+1} has been queued since p_i is still transmitting until $t'_i + \Delta t_i$ in the new schedule. With strict priority, p_{i+1} will start transmission right at $t'_i + \Delta t_i$ ignoring the IDLE packet. Hence, $t'_{i+1} = t'_i + \Delta t_i \leq t_i + \Delta t + \Delta t_i \leq (t_{i+1} + \Delta t)$.

Configuration	SRAM	TCAM	Metadata	Tbls	Regs
16×100 Gbps	80 KB	1.28 KB	85 b	3	1
32×25 Gbps	80 KB	1.28 KB	53 b	3	1

Table 3: Additional data plane resources for OrbWeaver’s weaved stream generation over an L2 forwarding switch. Ports are binned into groups of 2 and 4, and only 256 multicast groups reserved.

By induction, we have $t'_n \leq (t_n + \Delta t)$, that is, the latency impact is tightly bounded by Δt for an arbitrary user packet and won’t accumulate across packets. Given such fixed workload, consider the impact of the IDLE stream over the original transmission time $T = t_n + \Delta t_n - t_1$. For the new transmission time window $[t'_1, t'_n + \Delta t_n]$, the duration $T' = t'_n + \Delta t_n - t'_1 \leq \max(t'_n) + \Delta t_n - \min(t'_1) \leq t_n + \Delta t + \Delta t_n - t_1$. Hence, $T' - T \leq \Delta t$. Since $T \gg \Delta t$, the throughput of the high priority user packet stream is not impacted. \square

E Probability of Notification in Use Case #1

We can formally express the probability that a notification is sent before the flow is evicted. Consider the case where there is a drop in flow f and user packets are all MTU-sized, i.e., there is one packet per period, τ . Assume that the flow cache holds N records and 3 can be packed in each IDLE.

$$\begin{aligned} P(\text{notified}) &= \frac{P(\text{IDLE contains } f)}{P(\text{IDLE contains } f) + P(\text{new } f' \text{ replaces } f)} \\ &= \frac{\frac{3}{N}P(\text{IDLE})}{\frac{3}{N}P(\text{IDLE}) + \frac{1}{N}(1 - P(\text{IDLE}))P(\text{new flow})} \\ &= \frac{P(\text{IDLE})}{P(\text{IDLE}) + (1 - P(\text{IDLE}))P(\text{new flow})/3} \end{aligned}$$

where $P(\text{IDLE})$ is the probability that an IDLE packet was sent during a given period τ , and $P(\text{new flow})$ is the probability that a user packet’s flow cannot be found in the cache. Smaller packets multiply the second term in the denominator; a larger N decreases it by improving cache hit rates. The probability that a flow record is evicted *before* it is sent (i.e., that we miss the loss) is 1 less the above value.

F OrbWeaver Data Plane Resource Overhead

Section 3 details the overhead of OrbWeaver’s weaved stream generation on user traffic and energy usage. We note that OrbWeaver also uses data plane resources for IDLE seed packet filtering and replication, as shown in Table 3. For each category, OrbWeaver only occupies a small fraction of the total switch resources (for instance < 1% of both SRAM and TCAM).

CloudCluster: Unearthing the Functional Structure of a Cloud Service

Weiwei Pang

University of Southern California

Christophe Diot

Google Inc.

Sourav Panda

University of California, Riverside

Jehangir Amjad

Google Inc.

Ramesh Govindan

University of Southern California

Abstract

In their quest to provide customers with good tools to manage cloud services, cloud providers are hampered by having very little visibility into cloud service functionality; a provider often only knows where VMs of a service are placed, how the virtual networks are configured, how VMs are provisioned, and how VMs communicate with each other. In this paper, we show that, using the VM-to-VM traffic matrix, we can unearth the *functional structure* of a cloud service and use it to aid cloud service management. Leveraging the observation that cloud services use well-known design patterns for scaling (*e.g.*, replication, communication locality), we show that *clustering* the VM-to-VM traffic matrix yields the functional structure of the cloud service. Our clustering algorithm, CloudCluster, must overcome challenges imposed by scale (cloud services contain tens of thousands of VMs) and must be robust to orders-of-magnitude variability in traffic volume and measurement noise. To do this, CloudCluster uses a novel combination of feature scaling, dimensionality reduction, and hierarchical clustering to achieve clustering with over 92% homogeneity and completeness. We show that CloudCluster can be used to explore opportunities to reduce cost for customers, identify anomalous traffic and potential misconfigurations.

1 Introduction

As more online services migrate to the cloud, and as the user base of these services increases, the complexity and scale of cloud deployments has increased significantly. Today, cloud services routinely use tens of thousands of VMs, geographically dispersed for reliability and low-latency access to customers. Monitoring and managing a cloud deployment can be significantly challenging, since the performance, cost, and reliability of the service can depend on a large number of factors: how the cloud customer maps logical functionality to VMs, how the VMs are provisioned, where they are located, how well the paths between the VMs are provisioned, and so on. More generally, how well a cloud service works depends both on how well a customer designs the service, and how

well the provider provisions the underlying infrastructure.

Cloud service monitoring. Cloud providers struggle to provide customers with insights on the performance and reliability of a cloud service. This is because, while a VM provides a very convenient abstraction for computing and communication, the provider has (by design) very little *visibility* into cloud service logic embedded in the VM. This lack of visibility prevents providers from being able to relate problems observed at the service level to issues in the underlying infrastructure. For a given service, a provider often only knows where the VMs are, how much compute and storage each VM is provisioned with, customer-supplied names for the VMs, and how much traffic each VM exchanges with other VMs in the service. Customers are often loath to reveal more, for business and privacy reasons.¹

Today, major cloud providers (such as Amazon Web Services [12], Azure Cloud [2] and Google Cloud Platform [3]) provide customers with monitoring services. Their monitoring services (AWS CloudWatch [12], Azure Monitor [2] and Google Cloud Monitoring [3]) expose, using customizable dashboards, metrics capturing the state and activity of the cloud service’s VM instances (*e.g.*, their CPU and disk utilization, and the volume of network traffic to and from instances) as visible to the cloud provider, as well as other measures of the underlying networking infrastructure (*e.g.*, loss rates between instances). Some of these monitoring services also provide custom alerting mechanisms. Customers can define metrics that capture user-perceived performance, and configure alerts when these metrics exceed service-level objectives that cloud customers have with *their* customers.

Goal. Given that cloud service monitoring provides a competitive advantage, cloud providers continuously seek to add

¹**Ethical considerations:** For the 15 cloud projects we used in the evaluations in the paper (§4), we obtained explicit consent. For each project, we only used information available to the cloud provider: VM locations, VM names, and the VM-to-VM traffic matrix. We used the VM-to-VM traffic matrix to generate the clusters, and names and locations to evaluate the performance of CloudCluster. After our evaluations, we shared the results with each customer, and obtained feedback.

innovative capabilities to their monitoring systems, despite their limited visibility into cloud services. In this paper, we describe a new capability not, to our knowledge, previously considered in the literature: inferring the *functional structure* of a cloud service, *i.e.*, how a cloud service is modularized across its many VMs. Our work is inspired by a body of prior work on inferring structural relationships between components in a distributed system (§6).

To explain what we mean by functional structure, consider Figure 1(a) which shows the connectivity graph of VMs (which VMs communicate with which other VMs) of a cloud service. These VMs reside in different cloud regions (roughly, parts of a continent, see §2); most communication is within a region, but some communication exists across regions. With just the information that the cloud provider has, it can only obtain this kind of a view of the project. Now suppose that this cloud service is, in fact, architected as in Figure 1(b): it has a front-end load-balancer and a backend processing layer. With the information she has, the cloud service operator’s conceptual view of the structure of the service might be as shown in Figure 1(c): its VM instances are spread across two regions, with load balancer VMs (in red on the cluster on the left, and green on the cluster on the right) communicating with the processing-layer VMs (in magenta and cyan respectively) but not with other load balancer VMs, and the processing-layer VMs in each region communicating with each other as well. In addition, one of the load-balancers communicates with processing VMs in the other region (*e.g.*, due to overload in its own region).

The focus of our paper is to *unearth* the structure in Figure 1(c) *only using information available to the provider*. Specifically, we aim to develop methods that can extract this structure in which VMs are grouped into *VM groupings* by function and location. Ultimately, this will enable the provider to represent the service by a compact inter-grouping graph abstraction (Figure 1(d)).

In deriving the representations shown in Figure 1(c-d), CloudCluster can only determine that VMs in a cluster likely perform the same function, but cannot tell *which* function they perform (for example, whether the VMs run load-balancers, or image transcoders). This mitigates any privacy concerns cloud customers might have. Even so, we expect that in an actual deployment, a cloud provider will obtain consent from the customer before applying CloudCluster to the customer’s service.

Approach. We hypothesize that we should be able to infer VM groupings by *clustering the VM-to-VM traffic matrix of a cloud service*. Clustering a traffic matrix implies grouping together similar rows; two rows are similar if the traffic from their corresponding VMs to all other VMs is similar. Intuitively, two functionally similar VMs are likely to satisfy this property. For example, how two load-balancer VMs in a region are likely to communicate with all other processing layer VMs in the same region is likely to be similar, so clustering

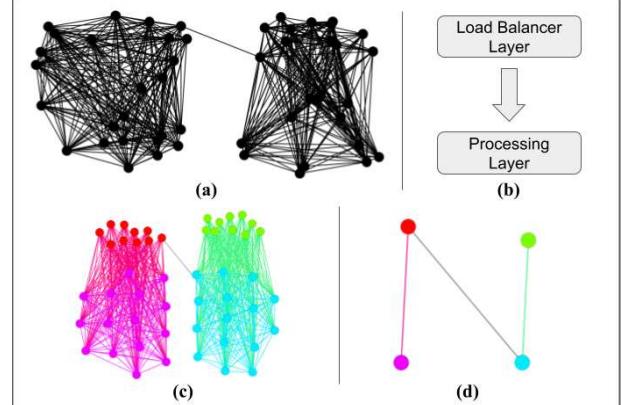


Figure 1: An example of different views of a cloud service: (a) VM connectivity graph as visible to the cloud provider; (b) The service architecture; (c) VM connectivity graph colored by function and location (the desired output of CloudCluster); (d) A compact inter-grouping graph abstraction.

will group them together. Furthermore, we expect clusters to be large because of the horizontal scaling employed by cloud services, which replicate processing or storage at a given layer using functionally identical VMs (*e.g.*, databases, in-memory stores, image transcoders *etc.*).

Challenge. Analyzing large VM-to-VM traffic matrices of real-world cloud services presents two challenges: scale, and robustness to variability and noise. At the scale of tens of thousand of VMs, any analysis must overcome the curse of dimensionality [60]; the sparsity of the traffic matrices in these higher-dimensions makes it difficult to derive insights from the data. Moreover, cloud services often vary in VM-to-VM traffic by several orders of magnitude, and methods of inferring their properties must accommodate this variability and be robust to noise introduced by the underlying measurement methodology (*e.g.*, by traffic sampling).

Contributions. This paper shows that *clustering* the VM-to-VM traffic matrix of a cloud service provider can help determine the functional organization of a cloud service, and that these clusters can be a useful abstraction for providing cloud customers with actionable insights into their service. To this end, the paper makes three contributions.

First, we develop a clustering algorithm, CloudCluster, that clusters VMs by similarity in their network communication characteristics (§3.4). CloudCluster is a novel combination of techniques, some known, and others new, to address the scaling and robustness challenges mentioned above. At its core, it uses a variant of *hierarchical clustering*, called agglomerative clustering [48] to determine clusters. This approach clusters VMs by proximity in some high-dimensional space. It requires a way to determine distance thresholds, and CloudCluster determines these thresholds dynamically in a data-driven manner. To scale better, it employs dimensionality reduction, and to be robust to variability in traffic volumes it scales traffic features (see §3 for more details).

Second, by evaluating the resulting clusters on 15 different

cloud service *projects*² (§4), we experimentally demonstrate that the resulting clusters *group together VMs by location and function*: *i.e.*, all VMs in a cluster are geographically co-located, and they perform the same function³. We verify this on cloud services that name VMs by function; for these projects, CloudCluster has homogeneity and completeness scores (metrics equivalent to precision and recall, respectively) of over 0.92 and 0.94 respectively.

Third, we demonstrate ways in which CloudCluster can be used to provide customers with actionable insights (§5). CloudCluster can analyze the inter-cluster graph (Figure 1(d)) to identify opportunities for reconfiguring VM placements to reduce cost: in one case, we found opportunities to reduce cost by 41.2% by provisioning an additional cluster to minimize inter-region traffic. It can also be used to detect anomalous traffic between clusters, to identify traffic shifts within a cloud project, or structural changes in the project across time. From 25 traffic anomalies reported either by an internal anomaly detector or the customer, a CloudCluster-based anomaly detector detected every anomaly, and identified the impacted clusters. CloudCluster can be used to detect potentially mis-labeled VM names (names that do not reflect function) or mis-provisioned VMs. In some projects, up to 1% of VMs appear to be mis-provisioned. In others, over 7% a project’s VMs appear to be mis-labeled — their traffic patterns differ from the majority of VMs that have the same labels.

2 Anatomy of a Cloud Service

In this section, we provide a brief background on the structure of cloud services. Our description focuses on Google’s cloud services; different service offerings may differ from this description in the details.

Google’s cloud resources are hosted in multiple locations worldwide. The network is subdivided into *regions* which are in turn divided into *zones* [9]. A region represents a part of a continent, and zones represent disjoint geographical areas within a region in which infrastructure resides. This partitioning permits cloud customers to coarsely control the placement of VMs to, for example, ensure low-latency access to customers, control cost and ensure high availability.

Customers can organize their cloud service into *projects* [4], which are granular functional groupings that simplify management of a cloud service. For example, an ad-supported social media service can have different projects for the user-facing front-end, the ads subsystem, and an analytics backend. Depending on the scale of the service, projects can be large, spanning tens of thousands of VMs across multiple regions. In this paper, we focus on the structure of projects.

VMs in a project are connected by one or more *virtual*

²As discussed in §2, a project is a granular functional grouping within a cloud service.

³In this paper, we use the term *function* to denote a long-lived heavy-weight service that forms part of a cloud service; we do *not* consider services deployed using ephemeral cloud functions (*e.g.*, lambdas).

networks [6] that provide isolation. Customers can organize VMs into *sub-networks* [7]: VMs in one sub-network must all be within the same region, and communicate over the same virtual network. Sub-networks simplify VM management tasks: *e.g.*, IP address assignment.

Customers populate projects with VMs. To create a VM, the customer: (a) selects a *configuration* for the VM (configurations differ in compute and storage), (b) specifies the VM’s *name* (the name is opaque to the provider, but customers may embed hierarchical structure into a name; some customers name VMs by function, a feature we leverage in evaluating CloudCluster in §4), (c) identifies the sub-network and the virtual network the VM uses, and (d) specifies the region and zone the VM is located in. This is the *only* information a cloud customer explicitly provides to Google. In addition, if customers opt in to flow logging [10], the logging service records VM-to-VM traffic for each enabled project.

3 CloudCluster Design

In this section, we describe CloudCluster, whose goal is to discover the underlying structure of a cloud project. We discuss how it scales to large cloud projects, while being robust to noise and variability.

3.1 Goals, Approach, and Overview

Notation. The input to CloudCluster is a VM-to-VM traffic matrix for a cloud project, containing traffic volumes between each VM over a fixed aggregation window.⁴ Traffic volumes are obtained by sampling flows. In §4, we discuss the actual values of the aggregation window and the sampling frequency. Formally, we denote this traffic matrix by \mathbf{Y} , with dimensions $n \times m$, where n is the number of source VMs (belonging to this specific project under consideration) and m is the number of destination VMs (which do not all have to belong to the same project, since VMs in a project can communicate with external clients or VMs in other projects).⁵ The i, j -th entry y_{ij} of \mathbf{Y} represents the volume of traffic (in bytes) from VM i to VM j , where $i \in [n], j \in [m]$.

Challenge: Noise. Since \mathbf{Y} is sampled, it is bound to be noisy. Aside from the error induced by sampling, measurement errors and randomness in traffic patterns can also induce noise. To model this, we can write:

$$\mathbf{Y} = \mathbf{M} + \mathbf{E} \quad (1)$$

⁴CloudCluster uses minimum possible aggregated information, namely the communication volume. Other metadata (*e.g.*, port numbers, process names) might be helpful in finding the functional structure. CloudCluster does not use this information. With consent from the customer, it might be possible to use this to improve our clustering, but we have left it to future work, in part because it is not clear whether customers will consent to revealing additional information.

⁵CloudCluster does not currently model traffic to cloud native services, like traffic to Google Cloud Storage [5]). Identifying traffic volumes from these services requires using other instrumentation services (*e.g.*, storage service logs), and we have left this to future work.

where \mathbf{M} is the unobserved noise-free, *true* traffic matrix and \mathbf{E} is a noise matrix. We assume e_{ij} is independent of all other entries and $\mathbb{E}[e_{ij}] = 0$ and $\text{Var}[e_{ij}] < \infty, \forall i \in [n], \forall j \in [m]$. This implies that $\mathbf{M} = \mathbb{E}[\mathbf{Y}]$.

Challenge: Scale. \mathbf{Y} can be large, since projects can have tens of thousands of VMs. We have observed, through manual inspection of cloud projects, that to enable projects to scale, designers often group VMs that perform similar functions. At the front-end, load-balancers redirect requests to VMs that scale with the request load; all these VMs perform the same function (*e.g.*, handle requests). In turn, at the back-end, these VMs may invoke other services that may be replicated across several identical VMs, or may send the request to a coordinator VM that invokes an iterative distributed computation spread across several identical VMs. Such structures result in *VM groupings*. We hypothesize that VMs in a group have similar traffic patterns (in terms of which VMs they communicate with, and the volume of traffic). If this hypothesis is true, then \mathbf{M} must be a *low rank matrix*.

We can formalize a VM grouping as:

Definition 1 VM Grouping. Let a VM Grouping be denoted by \mathbf{S}_i . Let m_u denote a row of matrix, \mathbf{M} . m_u belongs to \mathbf{S}_i , if

$$d(m_u, m_v) < \min_r \{d(m_u, m_r)\}, \forall v \in \mathbf{S}_i, \forall r \notin \mathbf{S}_i \\ \wedge d(m_u, m_v) < \delta, \forall v \in \mathbf{S}_i$$

$d(\cdot, \cdot)$ is some distance function and δ is a distance threshold.

Definition 1 implies that *similar* rows will be grouped together if they are most similar to each other and their similarity, quantified via a distance function, $d(\cdot, \cdot)$, is less than the distance threshold. In practice, this threshold can be different for different cloud projects.

Goal and Approach. Our goal is to discover all VM Groupings present in a cloud project. To do this, CloudCluster (a) estimates \mathbf{M} and then (b) clusters VMs (rows of \mathbf{M} with similar traffic patterns) to find the VM groupings.

To estimate \mathbf{M} , we leverage prior work, such as [27] and [21], which show that in a setting like ours, we can estimate well and with consistency the low-rank and noise-free, but unobserved, matrix, \mathbf{M} , from a random observation of the noisy matrix \mathbf{Y} , where $\mathbf{M} = \mathbb{E}[\mathbf{Y}]$.

Clustering algorithm: Overview. Using the estimate $\hat{\mathbf{M}}$, CloudCluster’s clustering algorithm⁶ seeks to extract VM Groupings according to Definition 1, with \mathbf{M} replaced by $\hat{\mathbf{M}}$. It must also address the scalability and robustness challenges identified above. To do this, CloudCluster’s algorithm has four components (Algorithm 1): 1) Feature scaling to transform the input traffic matrix. 2) Matrix estimation to estimate \mathbf{M} . 3) Hierarchical clustering to group similar VMs. 4) Cluster merging to fuse similar clusters. We describe each component in the following subsections.

⁶This is orthogonal to prior work that has explored clustering to group similar traffic matrices [59].

Algorithm 1: Steps in VM clustering

```

input :  $\mathbf{Y}$ , threshold  $\theta$  to merge similar clusters
output: Clusters merged_clusters
1 scaled_Y = feature_scaling(Y);
2 scaled_M_hat = TruncatedSVD(scaled_Y);
3 clusters = hierarchical_clustering(scaled_M_hat);
4 merged_clusters = merging(clusters, scaled_M_hat, theta);

```

3.2 Feature Scaling

What is a feature and why scaling is necessary. Each row of \mathbf{Y} can be treated as a (high-dimensional) feature. Then, identifying similarity in this feature space is equivalent to identifying VMs that have similar traffic patterns.

In practice, even within a single project, traffic volumes between VMs can span several orders of magnitude. This can make it difficult to discriminate between low and medium volume traffic patterns. Clustering relies on a distance metric, and many applicable distance metrics are disproportionately sensitive to larger values.

Log Scaling. Feature scaling normalizes the range of each feature to enable clustering algorithms to be robust to highly variable traffic volumes. Of the existing feature scaling methodologies, standardization and minmax scaling cannot handle the range of traffic volumes we see in cloud projects. Standardization replaces each feature’s value by how many standard deviations it is above or below the mean [1]. Minmax scaling transforms each individual feature value into the ratio between that value’s distance from the minimum to the range of values [1]. Traffic in cloud projects can span several orders of magnitude (from 0 to 10^9) and have skewed distributions; linear transformations like minmax scaling, or those that assume Gaussianity, like standardization, do not work well. For this reason, we choose *log scaling*, which uses the natural logarithm of the traffic instead of the original values; this handles volume variability much better (we demonstrate this experimentally in §4.4).

3.3 Estimating \mathbf{M}

Estimating singular values. Singular value thresholding can produce a good estimate, $\hat{\mathbf{M}}$, of the low-rank \mathbf{M} , using only observations from \mathbf{Y} (see [27], [21], [20]). However, estimating the number of singular values to keep cannot be determined exactly. After performing Singular Value Decomposition of the matrix, we choose the number of singular values to retain based on an *elbow* finding heuristic such as one introduced by [51]. The *elbow* suggests the approximate number of singular values to retain because most of the singular values after the *elbow* contribute little to the spectrum of the matrix. Figure 2 shows the spectrum of singular values for a traffic matrix for a project with over 3000 VMs. The sharp decline in the spectrum after about 50 singular values is indicative of a low-rank structure. Singular values in the tail which don’t quite decay to 0 indicate random noise (which tends to spread

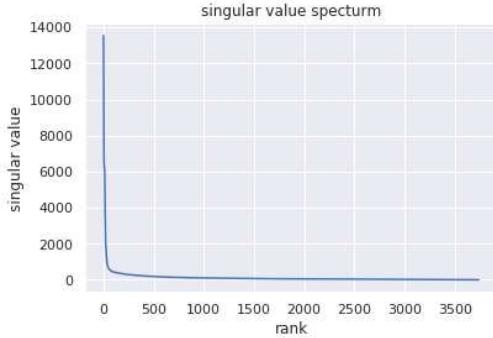


Figure 2: Singular Value spectrum of a traffic matrix with dimension (3742x3271)

across all orthogonal directions) with small finite variance (indicated by the small magnitude).

Extracting an r -rank approximation of \mathbf{M} . Once the number of singular values r is heuristically determined, performing an SVD produces the reduced rank estimate of the original matrix. Specifically, given the $n \times m$ original traffic matrix \mathbf{Y} , SVD produces the reduced dimension matrix $\hat{\mathbf{M}}$, such that:

$$\hat{\mathbf{M}} = \mathbf{U}_r \boldsymbol{\Sigma}_r \mathbf{V}_r^T,$$

where $\boldsymbol{\Sigma}_r$ is an $r \times r$ diagonal matrix of the singular values of \mathbf{Y} , \mathbf{U}_r and \mathbf{V}_r are orthonormal bases of dimensions $n \times r$ and $m \times r$, respectively. $\hat{\mathbf{M}}$ is a low-rank, *i.e.*, $\text{rank} = r \ll \min\{n, m\}$, approximation to the original matrix. However, $\hat{\mathbf{M}}$ is of dimensions $n \times m$. We need to project this matrix to an $n \times r$ subspace which will allow us to retain all the rows (associated individually to VMs), each of r -dimensional feature (columns). We denote this desired matrix by $\hat{\mathbf{M}}_r$, determined by:

$$\hat{\mathbf{M}}_r = \mathbf{U}_r \boldsymbol{\Sigma}_r$$

Effectively, the retained r singular values of the original matrix \mathbf{Y} determine how to scale each of the r -dimensional orthonormal vectors in \mathbf{U}_r . $\hat{\mathbf{M}}_r$ remains a good approximation of \mathbf{M} (in a reduced dimensional subspace) because it is simply a projection of each of the rows in $\hat{\mathbf{M}}$ (which is the best rank- r approximation of \mathbf{M}) on to a r -dimensional subspace. Both $\hat{\mathbf{M}}$ and $\hat{\mathbf{M}}_r$ are of rank r , and have the same norm.

The key benefit of SVD. Traffic matrices obtained from large cloud projects can have tens of thousand of rows and columns. The distance functions (used to compute row-similarity) scale exponentially in the number of features/columns. Moreover, in high dimensional feature spaces and with noisy data, distance metrics are unreliable [60] (the curse of dimensionality). Given this, a reduced-rank estimation of \mathbf{M} , and projection on to a feature-space of reduced dimensions allows our algorithm to remain robust to scale while retaining much of its structure.

3.4 Hierarchical Clustering

Infeasible clustering methods. Given the original matrix \mathbf{Y} , or the rank- r estimate $\hat{\mathbf{M}}_r$, we can use traditional clus-

tering techniques to find VM Groupings. For instance, prior approaches like [28] have established links between dimensionality reduction and K-Means clustering. However, for our use-case we would like to use dimensionality reduction for robustness to scale and noise but maintain fine control over the number of clusters to produce. Therefore, given that we do not know the number of clusters to produce, much of the existing work around K-Means [41] does not suffice for our needs. Density-based approaches such as DBSCAN [32] and OPTICS [22] do not require the number of clusters as input. However, they rely on other threshold parameters, estimating which requires domain knowledge (*e.g.*, information about a project beyond the sampled traffic volumes we have available) and maybe hard with high-dimensional data. MeanShift [29] and Affinity Propagation [33] also don't require the number of clusters, but their main drawback is time complexity, which depends on the number of iterations until convergence. We also show that MeanShift and Affinity Propagation don't perform well in the context of VM clustering in section 4.4.

Agglomerative clustering. Similar to density-based approaches, hierarchical clustering does not require the number of clusters *a priori*. CloudCluster uses agglomerative clustering [48], a bottom-up hierarchical clustering approach: each VM (row) starts in its own cluster, and clusters are recursively merged together. It uses Ward linkage [56] to determine which two clusters should be merged: at each iteration, this technique selects two clusters that minimize the increase in total within-cluster sum of squared error [44]. In the context of clustering VMs, doing this produces clusters of VMs with homogeneous traffic patterns, and this variance-minimizing property is similar to the K-Means objective function. The output of agglomerative clustering is a dendrogram (tree) of VMs (rows); the leaf nodes are the VMs (rows) and the non-leaf nodes are the nested clusters. Each non-leaf node has a value (“height”), which is the Ward distance [44] between the two entities merging at that node.

From hierarchical to flat clustering. In a dendrogram, each non-leaf node represents a potential cluster (containing all the leaf nodes in its sub-tree). CloudCluster must extract disjoint clusters from this dendrogram. To do this, it can use a static height threshold: each non-leaf node higher than this threshold is a distinct cluster. But, determining the threshold requires domain knowledge for each project. Instead, CloudCluster cuts the dendrogram based on *cluster inconsistency* [54]. For a given non-leaf node in the dendrogram with height h , if its sub-tree contains nodes with heights $H = \{h_0, h_1, \dots\}$, and mean of the heights is \bar{H} , and the standard deviation is $\sigma(H)$, the *inconsistency* of the node is: $inc = \frac{h - \bar{H}}{\sigma(H)}$.

When deciding whether to merge two sub-trees (or nested clusters), the inconsistency metric quantifies how different the new merged cluster would be compared to the nested clusters within it. A low value means that the merged cluster would be similar to the nested clusters under it. Conversely, a high

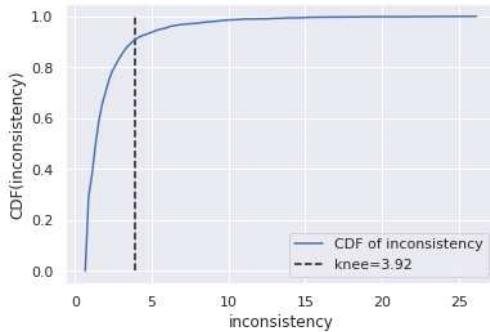


Figure 3: CDF of inconsistency value and the knee

inconsistency means that the merged cluster contains nested clusters which are fairly different. Therefore, the algorithm merges nested clusters when the inconsistency score is less than a threshold, μ .

Estimating μ . Instead of manually selecting the threshold μ , we use the following technique to estimate it. Closer to the leaves of the dendrogram, inconsistency values will be small. They will increase at non-leaf nodes higher in the dendrogram. For many projects, the distribution of inconsistency values is similar to Figure 3. This suggests that the *knee* of this curve is a good choice for μ because it identifies a transition between low and high inconsistency values. We use the knee locator implemented by [51] to determine μ . Then, we cut the dendrogram based on the threshold μ , resulting in a set of clusters.

3.5 Cluster Merging

In practice, we have found that our approach produces, for projects with thousands of VMs, tens or hundreds of clusters with small internal variation in terms of VM traffic patterns. However, it is too aggressive, and we find we can merge some of these clusters in a fast post-processing step. For this, we determine the centroid of each cluster produced by hierarchical clustering. Each centroid can be viewed as a *feature* of the candidate clusters. We treat each of these centroids as a new entity and cluster these entities. Inspired by MeanShift [29] which fuses clusters that are close to each other by comparing the distances to a threshold, we calculate the pairwise cosine distances of the clusters centroids and recursively merge pairs of clusters until no two clusters have a centroid distance less than a fixed merging threshold θ .

4 CloudCluster Evaluation

The goal of the evaluation is to demonstrate that CloudCluster produces clusters that are consistent with VMs grouped by location and function. In other words, *in each cluster, all VMs are in the same zone, and perform the same function*.

4.1 Methodology and Metrics

Dataset. We use anonymized, aggregated flow logs (specifically, Google VPC logs [10], please see footnote on page 1 for

a statement of the ethical use of customer data.) from cloud customers to generate our evaluation dataset. Our dataset includes projects ranging from a few thousand VMs to those with tens of thousands of VMs. We do not consider smaller (10-20 VMs) projects in our analysis; at these scales, less sophisticated tools can provide actionable insights. The dataset includes projects of VMs with various type of workloads (*e.g.*, web servers, load balancers, image transcoders, key-value stores *etc.*). It includes projects that are internal to Google and those belonging to external customers. Each traffic matrix in the dataset contains uniformly sampled VM-to-VM traffic aggregated over a 1-hour window. We use sampled data; sampling is necessary to scale measurement systems, and, as long as the sampling mechanism is uniform, we expect our clustering algorithm to work just as well as it would have on un-sampled data given that uniform sampling ought to preserve traffic volume relationships between VMs.

Implementation. Customer flow logs are stored in Google’s Colossus file system [30]. CloudCluster loads the flow logs into Dremel [43] and uses Dremel’s SQL-like queries to *select* data within the aggregation window, *group by* src-dst VM pairs and *aggregate* by volume. CloudCluster runs on a single VM with 128G memory, loads the aggregated result from Dremel into a dataframe, extracts the VM-to-VM traffic matrix, and then runs the algorithm described in §3. Traffic matrices for the projects we evaluate fit comfortably into a single VM.

Methodology. To demonstrate that CloudCluster produces clusters consistent with VMs grouped by location and function, we conduct two experiments on disjoint sets of the fifteen projects in our dataset:

i. Carefully-Named Group. The first experiment uses data belonging to eight projects. These eight projects (called the *Carefully-Named Group*) are different from the other seven projects *because we have information about the location and function of each VM*. For these projects, the customers have carefully named each VM based on function, likely to simplify manageability of the project. For example, VM naming schemes contain strings identifying well-known services (*e.g.* "redis" [25], "cassandra" [39], or "nginx" [53]). We call these strings *VM labels* (in addition to labels, VM names may contain, for example, instance identifiers). For projects in this group, we show that CloudCluster’s clusters, when further sub-grouped by the VM location (the VM’s zone, §2) match well with VM groupings by location and VM labels, *i.e.*, functions.

ii. Coarsely-Named Group. The second experiment uses data belonging to the remaining seven projects. For these, we have location information for each VM, but the VM naming scheme does not always indicate the function, or indicates function coarsely (we explain later precisely what this means). For this group of projects, we show that CloudCluster’s clusters, when further sub-grouped by the VM location, *do not*

match well with VM groupings by location (zone) and VM labels.

We emphasize that the cloud provider will always know a VM’s location, but cannot always know the VM’s function, since function-based naming is not a requirement of any cloud-service API that we are aware of.

Metrics. We use two standard measures of clustering goodness, *homogeneity* and *completeness* [49]. These are both scalar real-valued metrics in the range [0, 1]. In the context of VM clustering, *homogeneity* is the fraction of VMs in a same cluster that have the same location and VM label. Conversely, *completeness* is the fraction of VMs that have the same location and VM label that are in a single cluster. These are the analogs of precision and recall used in classification.

4.2 The Carefully-Named Group

The Carefully-Named Group refers to the eight projects where the VM’s are carefully named to reflect their function, in addition to the location (zone) information.

High homogeneity and completeness. We cluster the VMs in each of the eight projects in the Carefully-Named Group. As noted earlier, we further sub-group the clusters produced by location, *i.e.*, zone. Table 1’s third and fourth columns show the homogeneity and completeness for all the projects in this group. Across these projects, CloudCluster has high homogeneity: all projects have a homogeneity of 0.92 or higher, and for six of them the score is higher than 0.96. Completeness scores are also high: all projects have a completeness of 0.94 or higher. High completeness and homogeneity scores indicate good matching in the clustering results, and substantiate our central assertion: that CloudCluster’s clusters, when augmented with zone information, match VMs grouped by location and function.

What values of homogeneity and completeness are acceptable? Recall that these measures are the equivalent of precision and recall (respectively), for which acceptable thresholds depend upon the specific use case. Similarly, acceptable values of homogeneity and completeness depend upon what clustering is used for; we discuss this in §5.5. Also, as with precision and recall, we can trade-off homogeneity for completeness and vice versa; see §4.4 for an example.

CloudCluster works well for projects at different scales. Projects range in size from 500 VMs to over 10,000 VMs (second column of Table 1). The number of clusters (third column of Table 1) varies from a handful to around 200. CloudCluster also discovers clusters at different scales. Within project A, some clusters have more than 900 VMs, and some clusters have dozens of VMs or sometimes one. Moreover, CloudCluster can handle projects with varying functional and geographical diversity. Projects A, B, E and F each run more than 20 different kinds of software and span across a number of zones across the globe. This also explains why they have many clusters (recall that clusters are distinguished both by function and location). Projects C and D are functionally

homogeneous and scoped to a single continent; and projects G and H are moderately functionally diverse (5-6 different types of functions) but scoped within North America. This explains why C, D, G and H have only a handful of clusters.

Why location is important. Clustering groups VMs with a similar traffic pattern. Our hypothesis was that VMs that perform the same function will have similar traffic patterns. However, consider two VMs that perform the same function, but are located in zones on different continents. Although their traffic distributions to other VMs will be similar, they will likely send traffic to completely different sets of VMs (*e.g.*, load-balancers, other services) because they are located in different zones. Thus, their *rows* in the traffic matrix will be different, and CloudCluster will be unable to cluster them.

To illustrate the importance of location, for projects in the Carefully-Named Group, we compare completeness and homogeneity scores *without using location information*. This means that we no longer sub-group CloudCluster’s clusters by location (zone) *and* we do not use the location information when computing homogeneity and completeness scores. Table 1’s 5th and 6th columns show that, in this case, while homogeneity is reasonably high (all VMs in a cluster tend to have the same label, *i.e.*, *function*), completeness drops significantly for about half of the projects (*i.e.*, VMs with the same label do not all fall into the same cluster).

Label and cluster conflicts. Prior work has also explored a different way to characterize the quality of clustering [46]. Consider any pair of VMs. These VMs can either be in different clusters, or they can be in the same cluster. If clustering is perfect, then (a) if the VMs belong to different clusters, they must have different labels,⁷ and (b) if they belong to the same cluster, they must have the same labels. Conversely, clustering can fail in two ways: (a) the VMs belong to different clusters, but they have the same label (we call this a *cluster conflict*, which results in a completeness score lower than 1.0) and (b) the VMs belong to the same cluster, but have different labels (we call this a *label conflict*, which results in a homogeneity score less than 1.0).

To understand the magnitude of these conflicts, Table 1’s 7th and 8th column show the *rate* of cluster and label conflicts in each of our projects in the Carefully-Named Group. Following [46], we compute the rate of cluster conflicts as the fraction of all VM pairs in different clusters that have the same label, and the rate of label conflicts as the fraction of VM pairs in each cluster that have different labels. Table 1’s 7th and 8th column show that these numbers are negligibly small (less than half a percent) across all projects, and represents another way of viewing the results in Table 1’s 3rd and 4th column. For instance, for project D, label conflicts are zero, so its homogeneity is 1. Similarly, project C has high homogeneity because its label conflict rate is very small and

⁷More precisely, different labels or locations; we use labels to simplify the explanation

Project	#VM	#Cluster	CloudCluster w/ location info		CloudCluster w/o location info		Percentage of Conflict	
			Homogeneity	Completeness	Homogeneity	Completeness	Cluster conflict	Label conflict
A	10000+	72	0.984	0.966	0.942	0.312	0.020%	0.197%
B	5000+	206	0.919	0.951	0.888	0.706	0.046%	0.532%
C	500+	4	0.999	0.964	0.989	0.865	0.001%	0.614%
D	500+	3	1.000	0.938	0.989	0.831	0.000%	0.427%
E	5000+	60	0.966	0.940	0.929	0.901	0.212%	0.386%
F	5000+	177	0.937	0.949	0.873	0.929	0.127%	0.188%
G	5000+	8	0.996	0.971	0.992	0.971	0.001%	0.169%
H	1000+	6	0.997	0.997	0.997	0.997	0.000%	0.028%

Table 1: Homogeneity and completeness score (with and without location information) and percentage of conflict for projects in the Carefully-Named Group

project *H* has highest completeness and the lowest cluster conflict rate. (As an aside, these rates are defined on VM-pairs, so the actual rates cannot directly be matched to imperfections in homogeneity and completeness.).

Why CloudCluster is less than perfect. Given the diversity of project in the Carefully-Named Group, CloudCluster’s agreement with customer-provided functional groupings is impressive. However, it is less than perfect for several reasons.

Feature scaling compresses the range of each feature, which changes the relative feature distances of all VMs. Dimensionality reduction step removes information from all feature vectors. TruncatedSVD [34] only keeps the information of the specified number of dimensions. Merging might also induce errors. We use an approach similar to MeanShift’s postprocessing [29] in that we merge clusters that are similar to each others by a specified distance. Even though we choose a rather aggressive merging threshold, it is still possible to merge two groups of VMs that have different traffic patterns. Similarly, the merging threshold can also be so high that it breaks other sub-clusters which should be merged.

Finally, some of these label and cluster conflicts can be caused by inconsistently assigned VM labels. For instance, in project *F*, which has high homogeneity and completeness, we found some VMs labeled default or pool. Table 1 suggests that mis-naming of VMs in our Carefully-Named Group is small. As we discuss in §4.3, CloudCluster works less well for our Coarsely-Named Group because VM naming does not reflect function (*i.e.*, from the perspective of this analysis, the VMs are mis-labeled). Equally important, the non-zero rate of label and cluster conflicts suggests that, even for well-named projects, labels may be mis-configured; in §5.3 we discuss techniques to detect such misconfigurations.

4.3 Coarsely-Named Group

In §4.2, we showed that (a) CloudCluster has high homogeneity and completeness for projects where labels reflect functions, and VM location is taken into account, and (b) it has high homogeneity and low completeness when VM location is omitted. The Coarsely-Named Group contains projects where VM labels do not reflect function well. We ex-

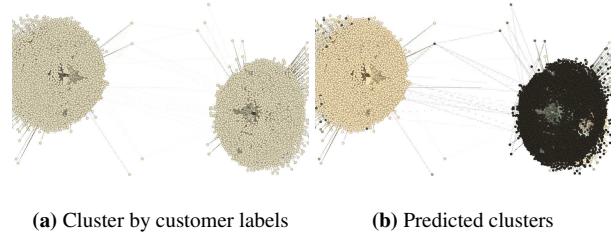


Figure 4: Same set of VMs clustered by (a) the customer label and (b) our algorithm. This figure shows VMs with generic labels like "default", "pool" or "farm".

pect CloudCluster to perform poorly in this case; we use this group of projects to rule out the possibility of other factors contributing to high completeness and homogeneity for the Carefully-Named Group.

As Table 2 shows, for projects I through O (which have comparable functional, size and spatial diversity as projects A-H), homogeneity is high, but completeness is low (for most projects in the 0.6-0.8 range, but in one case as low as 0.25). These results indicate that, in these projects VM labeling has the following property: if two VMs are similar in traffic characteristics, they are likely to have the same labels, but if they are different by traffic characteristics (so are in different clusters) they may still have the same labels. In other words, labels in these projects lack functional *specificity*.

Labeling specificity. Table 2 suggests that, if VM grouping by labels and location should match well with CloudCluster, labels have to have functional *specificity*. Some projects have less specific (or generic) functional labels, as we illustrate in the following examples.

Project	Homogeneity	Completeness
I	0.988	0.825
J	0.988	0.740
K	0.952	0.782
L	0.936	0.786
M	0.978	0.250
N	0.983	0.758
O	0.993	0.603

Table 2: Homogeneity and completeness scores for projects in the Coarsely-Named Group.

Figure 4 shows a group of VMs where the VM labels are generic (default, pool, or farm). In Figure 4(a), the dots are colored by the VM group they belong to by customer label. In this case, all VMs belong to the same group because they are assigned a generic label, so in Figure 4(a) they all have the same color (green). However, if we closely examine the functional structure of this project, we see two distinct groups densely connected internally, but sparsely connected externally. Figure 4(b) shows that CloudCluster is able to correctly distinguish between the two groups (yellow and black).

Sometimes, making labels specific enough requires careful thought. Figure 5 illustrates a case where *sharding* may require generic labels. In Figure 5(a), the customer has labeled all nodes within the circled ellipses as “loadbalancer”. However, from Figure 5(b), we observe that there is an internal structure to these load-balancers. They can be further separated into three groups where each has a distinctive connection pattern. The clustering algorithm captures this difference and puts them into different clusters.

Customers are not required to provide specific functional labels, but these examples give some insight into how CloudCluster’s clustering might differ from a customer’s notion of functional labels. At the same time, many projects *do* label VMs by function. For these, being able to identify generic customer labeling (or, more generally, mis-labeling) can help identify configuration errors (see §5).

4.4 Impact of Design Choices

We now quantify the importance of various design choices.

Dimensionality reduction. Dimensionality reduction reduces the runtime of the pipeline and improves the clustering accuracy. In the absence of dimensionality reduction, the distance metric can be unreliable for data with high-dimensional feature spaces [60]. Moreover, distance computation does not scale well for projects with more than 10,000 VMs; on our largest project, without dimensionality reduction, the pipeline takes more than **40 minutes** to finish. With dimensionality reduction, CloudCluster’s pipeline completed in **150 seconds** for the same project. CloudCluster is not latency-sensitive, but lower computational complexity is important for reducing the overhead or cost of executing CloudCluster’s algorithms

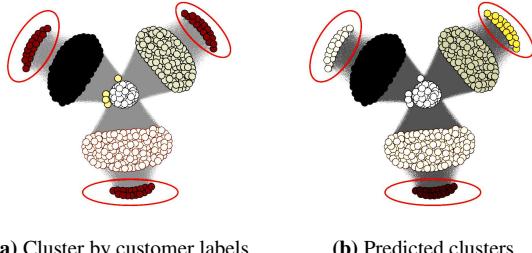


Figure 5: Same set of VMs clustered by (a) the customer and (b) our algorithm. This figure shows that customer-defined VM groups contains customer specific sharding.

on the cloud.

Feature scaling. Feature scaling approaches influence CloudCluster’s performance. If we disable feature scaling, CloudCluster produces lower homogeneity (0.812) and completeness (0.822) scores for project A, our largest project. By contrast, log-scaling is able to achieve 0.984 homogeneity and 0.966 completeness. Using other forms of feature scaling result in slightly lower homogeneity and completeness scores. Figure 3 shows that using standardized and minmax scaling reduces both homogeneity and completeness.

Hierarchical clustering. To validate our choice of our clustering algorithm, we compare with other plausible clustering approaches. We used OPTICS [22], Affinity Propagation [33] and MeanShift [29] to produce another set of clusters, and compared the clusters with project A’s labels. To be fair to these alternative clustering approaches, we performed the same feature scaling and dimensionality reduction before feeding the data into the algorithms. We used default parameters for these other clustering algorithms. We show that OPTICS results in significantly lower homogeneity (0.471) and completeness (0.163). Affinity Propagation produces slightly better homogeneity (0.994) by producing more than 3000 clusters in the project with 10,000+ VMs. This comes at the cost of a significantly lower completeness (0.559). Conversely, MeanShift achieves a higher completeness score (0.989) by having giant, noisy clusters, but with lower homogeneity (0.701).

Merging. Without merging, we achieve a slightly better homogeneity score (0.996), but a much worse completeness score (0.547). The high homogeneity is due to the fact that we produce clusters with small internal variation in the process of hierarchical clustering. The requirement of small internal variation divides VMs with similar traffic patterns into different clusters and lowers the completeness score. Merging combines similar clusters to significantly improve completeness for project A (from 0.547 to 0.966) at the expense of a small drop in homogeneity (from 0.996 to 0.984). This benefit of merging is evident across all projects in the Carefully-Named Group (Table 4). In some cases, the improvements in completeness are even more dramatic, increasing from 0.442 to 0.996 for project H.

	Homog.	Compl.
CloudCluster	0.984	0.966
Without feature scaling	0.812	0.822
Feature scaling: standardizer	0.939	0.953
Feature scaling: minmax scaler	0.974	0.948
Clustering: OPTICS [22]	0.471	0.163
Clustering: Affinity Prop [33]	0.994	0.559
Clustering: MeanShift [29]	0.701	0.989
Disable merging	0.996	0.547

Table 3: Compares the impact of different design choice on project A’s result

	Without Merging		With Merging	
	Homog.	Compl.	Homog.	Compl.
A	0.996	0.547	0.984	0.966
B	0.932	0.896	0.919	0.950
C	0.998	0.522	0.998	0.963
D	1.000	0.442	1.000	0.938
E	0.985	0.698	0.965	0.940
F	0.978	0.781	0.937	0.949
G	0.996	0.386	0.996	0.971
H	0.997	0.442	0.996	0.996

Table 4: Effect of merging on the Carefully-Named Group group

5 CloudCluster For Project Management

In this section, we describe several proof-of-concept ways in which the output of CloudCluster can help cloud providers provide their customers with actionable insights about the configuration and management of their services.

5.1 Reconfiguration to Reduce Cost

Cloud providers often price traffic in multiple tiers: traffic within the same cloud zone typically costs less than traffic between VMs from different zones, regions or continents. Customers engineer VM placements to reduce cost while balancing availability and proximity to customers. CloudCluster can help identify opportunities for reconfiguring VM placements to reduce costs. In this section, we discuss three examples that illustrate these opportunities; future work can develop systematic tools to discover such opportunities.

Figure 6 shows the distribution of traffic to other zones from VMs of project A belonging to VM label L . CloudCluster detects that VMs with this label belong to two different clusters: one which sends traffic more-or-less uniformly to VMs in 8 different zones (first cluster in Figure 6) and the other which sends over 80% of its traffic to a single zone (second cluster). A customer can potentially reconfigure the placement of VMs of the latter cluster to avoid inter-zone traffic. Although the traffic skew is visible across all VMs (so the customer might have been able to detect it using the VM label), CloudCluster is able to identify the precise set of VMs to re-configure.

Using CloudCluster, the cloud provider can determine the volume of intra-cluster and inter-cluster traffic, and determine how much of this traffic crosses zone, region, or continent boundaries. Using this, it can estimate cost savings resulting from reconfigured VM placements. Figure 7 and Figure 8 illustrate cost savings from reconfiguration in two cases.

The first case is a cluster C from project A of VMs located in different zones of a single cloud region. Almost 90% of traffic in C is intra-zone, which is free or relatively cheap on most cloud providers ([15], [8], [11]). However, the remaining traffic traverses continental boundaries, and accounts for a significant fraction of total cost charged to C . If the customer were to provision a small cluster in the zone on the other continent where the traffic comes from, it can reduce the cost attributable to this cluster by 41.2% (Figure 7).

The second case is a cluster C' of project A whose traffic is largely inter-region (intra-zone traffic is < 0.1%). 92.3% of egress traffic from C' goes to zones in another region R , and 95.9% of its ingress traffic comes from VMs in a single zone in R . Moving VMs in C' to R (an *egress-favored* placement) reduces cost by 21.1%, while moving these VMs to the zone in R from which they receive most traffic (an *ingress-favored* placement) reduces cost by 15.1% (Figure 8).

These are simplified examples; in practice, tools that suggest re-configuration of VM placements will need to consider other customer objectives such as availability and latency. We have left development of such tools to future work, but CloudCluster’s clustering can be a valuable input to such tools.

5.2 Anomaly Detection

Large-scale cloud project outages are sometimes caused by rapid increases in service workload, management operations by the customer (incorrect service configuration), by the provider (VM migration), or failures in the provider’s network. These are often accompanied by sudden shifts in traffic between VMs in the service or traffic to and from external entities (e.g., customers of the cloud service). Such traffic shifts may often be visible in the aggregate traffic between clusters. Because our clusters correspond to functionally homogeneous VMs, if one VM in cluster A starts communicating more with a VM in cluster B , it is likely that all other VMs in A will also start communicating more with VMs in B .

In this section, we present a preliminary evaluation of an anomaly detector that tracks significant deviations in aggregate inter-cluster traffic on each link in the inter-cluster graph (Figure 1(e)). Such a detector can also help localize anomalies, as we discuss below. In practice, we expect our anomaly detector to complement other approaches used by cloud providers.

The anomaly detector works as follows. For each edge in the inter-cluster graph (an edge exists between two clusters if their VMs communicate), it tracks at each aggregation window, the total volume in bytes, the total flow count, and the number of communicating VM pairs between each pair of clusters. When, for a given edge, any of these quantities deviates significantly from a windowed moving median [57], we flag that deviation as an anomaly (we omit the details for brevity). Because the inter-cluster graph is sparser than the inter-VM graph (e.g., Figure 1(a)), we are able to scalably identify correlated anomalies, where two or more communicating cluster pairs exhibit anomalous traffic at the same time.

Trace Analysis and Results. To quantify the effectiveness of this detector, we identified 25 time windows across different projects where either (a) an internal anomaly detector that uses a different methodology flagged anomalous traffic in the project during the corresponding time window (17 instances) or (b) the customer filed a trouble ticket (8 instances).

We then ran the CloudCluster-based anomaly detector on these 25 time windows, and, in each case, were able to con-

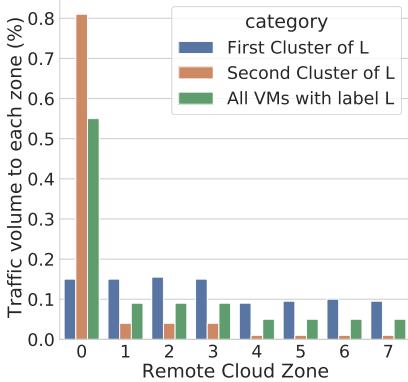


Figure 6: CloudCluster finds VMs with same label but different traffic patterns.

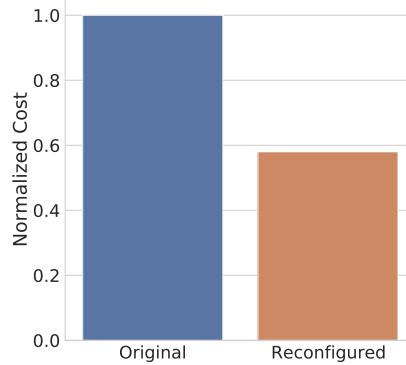


Figure 7: Original vs. Reconfigured placement cost

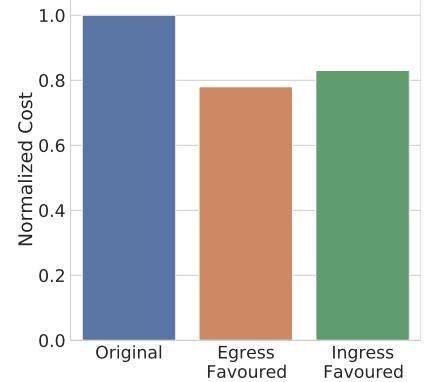


Figure 8: Original vs. Egress-favoured vs. Ingress-favoured placement cost

firm the existence of the anomaly⁸, and also to pinpoint which cluster-pairs were responsible for the deviations. We have not analyzed false positive rates; since we started with known anomalies flagged by other systems. For the 8 customer-reported incidents, our detector was able to correctly identify the offending cluster-pairs (as determined in the post-mortem reports). We identified two broad classes of anomalies: *traffic shifts* and *structural changes*. In the first class, the inter-cluster graph does not change, but traffic on some subset of links changes significantly. In the second, new nodes and/or edges are added to the graph or nodes and edges are removed. Of the 25, three were traffic shifts and the rest were structural changes.⁹

Our detector is fast: the maximum processing latency to compute the deviation scores, across all projects, was 92.3 milliseconds per time window.

The following paragraphs briefly describe some qualitatively different anomalies that we were able to detect; §A contains a more detailed description.

Correlated traffic shift due to peering router failure. This anomaly was reported by the network operator in reaction to a peering router failure. Our detector observed that a cluster in the region nearest the peering router saw a sudden reduction in flow and byte counts. Concurrently, a cluster in another region, (which, from label names, we determined was functionally identical to the first cluster), saw an increase in traffic. We suspect that the peering router failure diverted external traffic to enter the cloud provider’s network at a different location, but don’t have the instrumentation to confirm this.

Structural change due to VM migration. This anomaly was reported by the internal anomaly detector. Our CloudCluster-based anomaly detector identified a sequence of structural changes across successive aggregation windows. Recall that clusters are distinguished both by function and lo-

cation (§4.2). In this case, the structural changes were caused by a migration of VMs from one server to another due to scheduler-driven evictions. The migration was spread out over multiple aggregation windows, so our detector noticed a sequence of structural changes corresponding to progressive migration of VMs from one server to another.

Structural change due to project reconfiguration. Our internal anomaly detector flagged anomalous traffic for a cloud provider. The CloudCluster-based anomaly detector identified a structural change: two clusters were removed from the graph and one was added. The two initial clusters corresponded to a singleton cluster containing a leader VM and another containing 120 worker VMs. The new cluster contained the 121 VMs, encompassing both the leader and the workers. In this case, it turns out that the customer had initiated the structural change, decommissioning the older VMs in favor of another set of VMs as part of an upgrade.

5.3 Potential Label Misconfiguration

As discussed in §4.2, several customers label VMs with precise function names and location information. We conjecture that they use this to simplify project management. These VM labels are often configured, either by hand or by a script. *Label misconfigurations* can occur, and CloudCluster can be used to detect the *likely* candidates. When a label misconfiguration occurs in a project whose VMs appear to be named by function and location, *i.e.*, when the project has a high homogeneity and completeness, it manifests either as a label conflict or a cluster conflict (§4.2).

Cluster Conflict. A cluster conflict occurs when VMs belong to different clusters, but have the same labels. Such a conflict can either result from a misconfigured label, or from a clustering error. To distinguish between those two cases, we use a technique inspired by prior work in clustering on *silhouette analysis* [50], which attempts to measure the intrinsic performance of clustering. This analysis assigns each item (or VM, in our context) a score in the range $[-1, 1]$ that measures how similar the VM is to its own cluster, compared to other

⁸A more detailed analysis of the detector performance, and comparisons with other detection techniques, is beyond the scope of this paper.

⁹Some of these structural changes or traffic shifts might be intentional, even though our approach flags them as (statistical) anomalies.

clusters.

We modified this idea to derive a metric that quantifies whether a customer label is too generic (*i.e.*, spans multiple clusters) or too specific. Let V_l be the set of VMs that has a customer-defined label l , but CloudCluster splits it up into n clusters $\{C_1, C_2, \dots, C_n\}$. Let \bar{c}_l be the centroid, in feature space, of the traffic features of all VMs in V_l . Let \bar{c}_i be the centroid of the traffic features of all VMs in C_i (the C_i s might contain VMs not in V_l). For each cluster C_i , let a_i be the average distance of each VM in this cluster to \bar{c}_l and b_i be the average distance of each VM in this cluster to \bar{c}_i . Then, consider the following metric: $ms(i) = \frac{b(i)-a(i)}{\max(a(i), b(i))}$.

Intuitively, if $ms(i) < 0$, each VM in the cluster is closer to the cluster center than to the label's center, so the labeling is too generic. Conversely, if $ms(i) > 0$, then the label is too specific. Either way, this indicates a mismatch between clustering and customer-provided labeling, which can be used in some cases to identify potential mis-labeled VMs.

To detect mis-labeling VMs using this technique, we apply the following algorithm. Without loss of generality, assume that C_1 has the largest number of VMs with label l . For all $i > 1$, if $ms(i) < \psi$ (a conservative threshold < 0 , we use -0.5), we mark all VMs in C_i with label l as mis-named.

The output of this analysis is a list of *potentially* (we use this term to indicate that, ultimately, any such mis-labeling would have been verified by a customer, since the customer understands the *intent* behind the naming) mis-labeled VMs that cause cluster conflicts.

Table 5 lists the fraction of potentially mis-labeled VMs for four of our projects. These four projects belonged to a customer who gave us feedback on our clustering results. The fraction of mis-labeled VMs range from negligible amounts (*e.g.*, project H has 0.1% mis-named VMs) to a few percent (for projects A , E and F). For these projects, we were able to verify with the customer that our identification of mis-labeled VMs was accurate. In these cases, the customer had changed the functions in some VMs but forgot to update the VM labels.

Label conflicts. Mis-labeling can also cause label conflicts: different labels within the same cluster. Table 5 also shows the rate of occurrence of these. They happen less frequently, and often fall into two categories. VMs labeled generically such as “default” fall into the same cluster as VMs with more specific labels (*e.g.*, “app-server”). A second cause of mis-labeling is inconsistent hyphenation (*e.g.*, “appserver” vs. “app-server”), or inconsistent abbreviations (*e.g.*, using “es” instead of “east”). We identified examples in the second category using manual inspection; future work can automate the detection of mis-labeling in this category using edit-distance based string similarity analysis [55].

5.4 Potentially Mis-provisioned VMs

When configuring a VM, project owners can provision VM resources by specifying the *machine type* for each VM. Machine types determine the capacity of the VM instances

Project	Cluster Conflict (%)	Label Conflict (%)	Mis-provisioning Rate (%)
A	4.62	0	1.59
E	5.75	3.15	0
F	7.26	0.10	0.80
H	0.09	0.04	0.38

Table 5: The percentage of VMs that are mis-labeled in each project (§5.3), the rate of misprovisioning (§5.4).

in terms of CPU cores, memory and egress network bandwidth [35]. Different machine types are priced differently, so over-provisioning a VM can have cost implications. Misprovisioning can also impact performance: under-provisioned VMs can result in stragglers, causing services to violate their latency SLOs.

CloudCluster can identify mis-provisioned VMs by determining outlier machine types in a cluster. Since CloudCluster’s clusters identify VMs performing a similar function, if most VMs in a cluster are of machine type a , but a small number are of machine type b , we can identify the latter set as mis-labeled VMs. In determining the rate of mis-labeling, we must filter out mis-labeled VMs. To be more robust to clustering errors, we flag a VM as mis-labeled if it does not lie at the edge of the cluster (as determined by distance from the cluster centroid in feature space).

Table 5 shows the rate of mis-provisioning in 4 of the projects in the Carefully-Named Group. A small number, 1%, appear to be mis-provisioned. We say “appear to be” because the operator cannot know the intent of the customer; they may have deliberately provisioned these machines differently to run additional tasks (*e.g.*, compute bound jobs whose footprint is not visible in the VM-to-VM traffic matrix). Any mis-provisioning will ultimately have to be verified as such by manual inspection by the customer.

5.5 Discussion

In §4.2, we said that acceptable values of homogeneity and completeness depend upon what clustering is used for. We conclude this section with a brief qualitative discussion of this issue, leaving quantitative analysis to future work.

We have described two types of use cases in this section. Reconfiguration and anomaly detection are based upon the inter-cluster graph abstraction, and specifically upon inter-cluster traffic volumes. For these cases, if *most* VMs (*e.g.*, 90%) in a cluster are functionally similar, the reconfiguration decision, or the anomaly detection is likely to be correct.

Detecting mis-labeled or mis-provisioned VMs requires comparing attributes of VMs within a cluster. This can be more susceptible to false positives and false negatives, unless homogeneity and completeness are very high. Because a cloud provider cannot always know the homogeneity and completeness *a priori*, using clustering for these tasks requires additional filtering steps to minimize false positives. For mis-provisioned VMs, we filter candidates at the edge of the cluster (§5.4). For mis-labeling, we use silhouette analysis (§5.3).

6 Related Work

Inferring Structure from Traffic. Complementary to CloudCluster, others have explored inferring host behavior and distributed system properties from network traffic. The closest prior work [58] groups Internet hosts within each IP prefix by traffic similarity, and explores how this can be used to detect malicious behavior. Other work has modeled host-to-host communication as a graph to understand properties of inter-host communication [13, 14, 36], to infer botnet structure [45], or the logical structure of enterprise networks [16]. The body of work on tracing in distributed systems seeks to infer the causal structure as well as other properties of distributed systems from RPC traces to aid performance debugging (*e.g.*, [19, 47, 52]). Other work has used traffic to infer specific characteristics of VMs in cloud settings: strongly connected groups of VMs as candidates for migration [26], or compromised VMs [23]. Some of these use clustering [26, 58], but do not consider scale and robustness to range of traffic volumes.

Data Clustering. Clustering is a mature area of research, with many established techniques such as K-Means [41], DBSCAN [32], OPTICS [22], AffinityPropogations [33], Hierarchical Clustering [48], *etc.* That clustering is susceptible to the curse of dimensionality is well-known [60]. Clustering in high dimensions has been explored extensively either by: (a) using heuristics to determine attributes of sub-spaces (*e.g.*, CLIQUE [18] or SUBCLU [37]) or (b) designing special distance measures (*e.g.*, projected clustering, as in PreDeCon [24] or PROCLUS [17]). In contrast, CloudCluster explicitly reduces the dimension of the VM-to-VM traffic to the point where conventional clustering techniques and similarity measures are applicable.

Cloud Monitoring and Workload Characterization. Tangentially relevant prior work has used CPU and memory utilization traces to infer properties of VMs [31, 38, 42].

7 Conclusion

CloudCluster performs clustering on the VM-to-VM traffic of cloud projects and yields the functional structure of the cloud service. It overcomes the challenges imposed by scale (cloud services contain tens of thousands of VMs), by orders-of-magnitude variability in traffic volume and measurement noise, and by the lack of prior knowledge of the cloud projects (for number of clusters). The output of CloudCluster can help detect potentially mis-provisioned or mis-labeled VMs, identify opportunities to reduce cost, and detect anomalies.

Future work. Several directions of future work remain, including: identifying the frequency at which to apply CloudCluster to projects; incrementally adjusting clusters when VMs leave or join; supporting traffic to cloud native services such as storage; exploring better methods for determining the cluster merging threshold; more thoroughly evaluating the accuracy of cost reconfiguration, anomaly detection, or miscon-

figuration determination, and comparing their performance against other alternatives; determining whether additional information from customers, obtained with their consent, can improve the quality of the resulting functional structure.

References

- [1] 6.3. preprocessing data. <https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing>.
- [2] Azure monitor overview - azure monitor. <https://docs.microsoft.com/en-us/azure/azure-monitor/overview>.
- [3] Cloud monitoring | google cloud. <https://cloud.google.com/monitoring>.
- [4] Creating and managing projects. <https://cloud.google.com/resource-manager/docs/creating-managing-projects>.
- [5] Google cloud storage. <https://cloud.google.com/storage>.
- [6] Google vpc. <https://cloud.google.com/vpc/docs>.
- [7] Network and subnetwork terminology. https://cloud.google.com/vpc/docs/vpc#subnets_vs_subnetworks.
- [8] Network pricing|compute engine documentation|google cloud. <https://cloud.google.com/compute/network-pricing>.
- [9] Regions and zones | compute engine documentation | google cloud. <https://cloud.google.com/compute/docs/regions-zones>.
- [10] Using vpc flow logs. <https://cloud.google.com/vpc/docs/using-flow-logs>.
- [11] Virtual network pricing: Microsoft azure. <https://azure.microsoft.com/en-us/pricing/details/virtual-network/>.
- [12] Clouds project cloudwatch. <https://aws.amazon.com/cloudwatch/>, 2000.
- [13] Blinc. *ACM SIGCOMM Computer Communication Review*, 35(4):229–240, 2005.
- [14] Network monitoring using traffic dispersion graphs (TDGs). *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, (c):315–320, 2007.
- [15] Aws site-to-site vpn and accelerated site-to-site vpn connection pricing. <https://aws.amazon.com/vpn/pricing/>, 2020.
- [16] Role classification of hosts within enterprise networks based on connection patterns. *Proceedings of the General Track: 2003 USENIX Annual Technical Conference*, pages 15–28, 2003.

- [17] Charu C Aggarwal, Joel L Wolf, Philip S Yu, Cecilia Procopiuc, and Jong Soo Park. Fast algorithms for projected clustering. *ACM SIGMod Record*, 28(2):61–72, 1999.
- [18] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data. *Data Mining and Knowledge Discovery*, 11(1):5–33, 2005.
- [19] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP ’03, page 74–89, New York, NY, USA, 2003. Association for Computing Machinery.
- [20] Muhammad Amjad, Vishal Misra, Devavrat Shah, and Dennis Shen. Mrsc: Multi-dimensional robust synthetic control. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(2), June 2019.
- [21] Muhammad Amjad, Devavrat Shah, and Dennis Shen. Robust synthetic control. *Journal of Machine Learning Research*, 19(22):1–51, 2018.
- [22] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod record*, volume 28, pages 49–60. ACM, 1999.
- [23] Behnaz Arzani, Selim Ciraci, Stefan Saroiu, Alec Wolman, Jack Stokes, Geoff Outhred, and Lechao Diwu. Privateye: Scalable and privacy-preserving compromise detection in the cloud. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 797–815, Santa Clara, CA, February 2020. USENIX Association.
- [24] Christian Bohm, K Railing, H-P Kriegel, and Peer Kroger. Density connected clustering with local subspace preferences. In *Fourth IEEE International Conference on Data Mining (ICDM’04)*, pages 27–34. IEEE, 2004.
- [25] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., USA, 2013.
- [26] Marco Cello, Kang Xi, Jonathan H Chao, and Mario Marchese. Traffic-aware clustering and vm migration in distributed data center. In *Proceedings of the 2014 ACM SIGCOMM workshop on Distributed cloud computing*, pages 41–42, 2014.
- [27] Sourav Chatterjee. Matrix estimation by universal singular value thresholding. *The Annals of Statistics*, 43(1):177–214, Feb 2015.
- [28] Michael B. Cohen, Sam Elder, Cameron Musco, Christopher Musco, and Madalina Persu. Dimensionality reduction for k-means clustering and low rank approximation. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC ’15, page 163–172, New York, NY, USA, 2015. Association for Computing Machinery.
- [29] D. Comaniciu and P. Meer. Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619, May 2002.
- [30] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google’s globally-distributed database. In *OSDI*, 2012.
- [31] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP ’17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery.
- [32] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD’96, page 226–231. AAAI Press, 1996.
- [33] Brendan J. Frey and Delbert Dueck. Clustering by passing messages between data points. *Science*, 315(5814):972–976, 2007.
- [34] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions, 2009.
- [35] Google Inc. Machine types | compute engine documentation | google cloud. <https://cloud.google.com/compute/docs/machine-types>.
- [36] Yu Jin, Esam Sharafuddin, and Zhi-Li Zhang. Unveiling core network-wide communication patterns through application traffic activity graph decomposition. *SIGMETRICS Perform. Eval. Rev.*, 37(1):49–60, June 2009.

- [37] Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. Density-connected subspace clustering for high-dimensional data. In *Proceedings of the 2004 SIAM international conference on data mining*, pages 246–256. SIAM, 2004.
- [38] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *2012 IEEE Network Operations and Management Symposium*, pages 1287–1294. IEEE, 2012.
- [39] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.
- [40] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, 49(4):764–766, 2013.
- [41] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.
- [42] Shruti Mahambre, Purushottam Kulkarni, Umesh Bellur, Girish Chafle, and Deepak Deshpande. Workload characterization for capacity planning and performance management in iaas cloud. In *2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 1–7. IEEE, 2012.
- [43] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1–2):330–339, September 2010.
- [44] Fionn Murtagh and Pierre Legendre. Ward’s hierarchical clustering method: clustering criterion and agglomerative algorithm. *arXiv preprint arXiv:1111.6285*, 2011.
- [45] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. Botgrep: Finding p2p bots with structured graph analysis. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security’10, page 7, USA, 2010. USENIX Association.
- [46] William M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.
- [47] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. WAP5: Black-box performance debugging for wide-area systems. *Proceedings of the 15th International Conference on World Wide Web*, pages 347–356, 2006.
- [48] Lior Rokach and Oded Maimon. *Clustering Methods*, pages 321–352. Springer US, Boston, MA, 2005.
- [49] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*, pages 410–420, 2007.
- [50] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53 – 65, 1987.
- [51] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a "kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st international conference on distributed computing systems workshops*, pages 166–171. IEEE, 2011.
- [52] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.
- [53] Igor Sysoev et al. Nginx. Inc., “nginx,” <https://www.nginx.com>, 2004.
- [54] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0—Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints*, page arXiv:1907.10121, Jul 2019.
- [55] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.
- [56] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

- [57] Eric W Weisstein. Moving median. <https://mathworld.wolfram.com/MovingMedian.html>.
- [58] Kuai Xu, Feng Wang, and Lin Gu. Network-aware behavior clustering of internet end hosts. In *2011 Proceedings IEEE INFOCOM*, pages 2078–2086. IEEE, 2011.
- [59] Y. Zhang and Z. Ge. Finding critical traffic matrices. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 188–197, 2005.
- [60] Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. A survey on unsupervised outlier detection in high-dimensional numerical data. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 5(5):363–387, 2012.

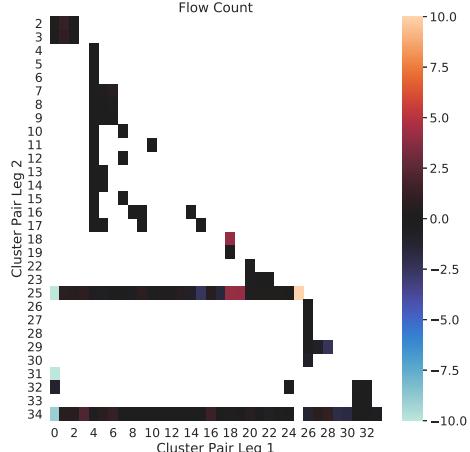


Figure 9: Cluster to Cluster flow count deviation scores

Appendix

A Detailed Explanation of Anomalies

Correlated traffic shift due to peering router failure. This anomaly was reported by the network operator in reaction to a peering router failure. Our detector observed that a cluster in the region nearest the peering router saw a sudden reduction in flow and byte counts. Concurrently, a cluster in another region, (which, from label names, we determined was functionally identical to the first cluster), saw an increase in traffic. We suspect that the peering router failure diverted external traffic to enter the cloud provider’s network at a different location, but don’t have the instrumentation to confirm this. Figure 9 depicts the cluster to cluster deviations scores expressed in terms of median absolute deviations (MAD) [40] with the sign indicating whether the upper (e.g. positive) or lower (e.g. negative) anomaly detection boundary was crossed. The x-axis and y-axis represent either the source or destination of the cluster pairs whose interactions are studied. The values of the heat map show the MAD deviations observed between the interacting cluster pair and is assigned a color based on the color map where the color black indicates no deviation and warmer (calmer) colors represent anomalous traffic characteristic deviating above (below) the median traffic volume. In the observed traffic shift, we were quickly able to identify the cluster pairs impacted by the anomaly (e.g. $|dev| > 5$ in Figure 9) that appear as the red, orange, and blue cells in the heatmap, filter away clusters not impacted by the anomaly that appear as black cells in the heatmap, and provide a project wide summary of the anomaly.

Structural change due to VM migrations. This anomaly was reported by the internal anomaly detector. Our CloudCluster-based anomaly detector identified a sequence of structural changes across successive aggregation window. Recall that clusters are distinguished both by function and location (§4.2). In this case, the structural changes were caused by a migration of VMs from one server to another due to scheduler-driven evictions. The migration was spread out

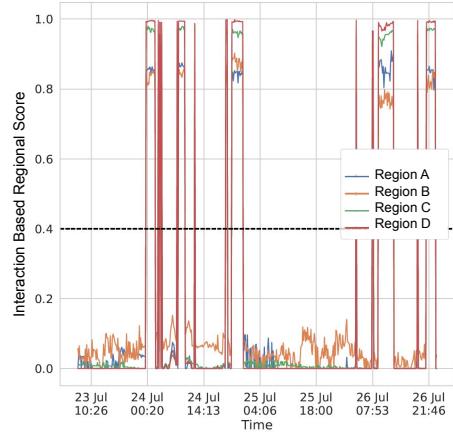


Figure 10: Regional anomaly scores during VM eviction/migration

over multiple aggregation window, so our detector noticed a sequence of structural changes corresponding to progressive migration of VMs from one server to another. Figure 10, shows the regional-score, an average of all the cluster to cluster deviation scores weighted by the number of VM communication pairs, computed for every region. The recurring structural changes manifest as plateaus and valleys in the regional score observed. The valleys represent the time windows where the new cluster behaviour is learnt and clusters behave as *normal*, while the plateau’s illustrate the time windows where there is a migration storm (e.g. significant number of VM migrations).

Load increase. A customer reported, to the cloud provider, a trouble ticket asking to root cause a missed service-level agreement. The CloudCluster-based anomaly detector identified a sudden increase of external traffic to clusters with memcached servers in one of the customer’s projects. (CloudCluster models external traffic as a single node in the inter-cluster graph). This was also concluded in the manual postmortem of the event. Similar to Figure 9 where we characterize the deviation in cluster pair interactions, here we observed a drift in interactions against logical clusters (e.g. may not contain VMs such as client IP) representing connections external to the project. Using this, we identified the culprit traffic flows through heavy hitter analysis and their origin, which happens to be a project that auto scaled to keep up with traffic demand and subsequently flooded the memcached projects with requests.

Structural change due to project reconfiguration. Our internal anomaly detector flagged anomalous traffic for a cloud provider. The CloudCluster-based anomaly detector identified a structural change: two clusters were removed from the graph and one was added. The two initial clusters corresponded to a singleton cluster containing a master VM and another containing 120 worker VMs. The new cluster contained the 121 VMs, encompassing both the master and the workers. In this case, it turns out that the customer had initiated the structural

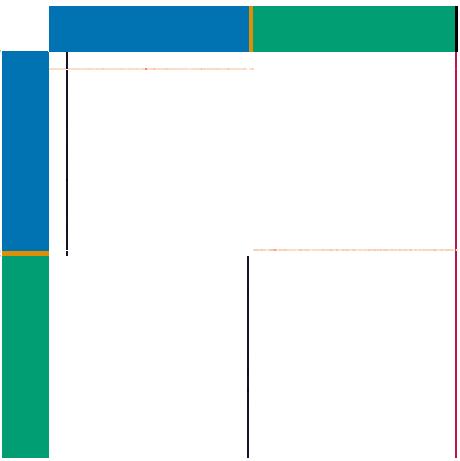


Figure 11: VMxVM Communication Graph (grouped by cluster assignment)

change, decommissioning the older VMs in favor of another set of VMs as part of an upgrade. Figure 11 shows the project communication structure using a VMxVM matrix where the values of the heatmap show the number of bytes sent between VMs log scaled (e.g. white represents no communication and warmer colors depict higher bandwidth consumption). The VMs in the rows and columns are sorted by their cluster assignments wherein they are grouped and identified by the color strip (e.g. cluster id) that appear on the top and left side of the heatmap (e.g. blue, yellow and green). On visualizing the project structure using this heatmap, the aforementioned project re-instantiation evolves in the following manner: a) Initially the top-left sub-structure (e.g. one master, 120 workers) operated devoid of the bottom-right substructure. b) After a downtime where the cluster-to-cluster deviation scores exceed a predefined threshold, we observed the bottom-right structure, which had displaced the other sub-structure. Therefore, by analyzing the traffic patterns, the network-engineer can identify changes to the project structure.

Zeta: A Scalable and Robust East-West Communication Framework in Large-Scale Clouds

Qianyu Zhang¹, Gongming Zhao^{1*}, Hongli Xu^{1*}, Zhuolong Yu², Liguang Xie³

Yangming Zhao¹, Chunming Qiao⁴, Ying Xiong³, Liusheng Huang¹

¹*University of Science and Technology of China,*

²*Johns Hopkins University*, ³*Futurewei Technologies*, ⁴*SUNY at Buffalo*

Abstract

With the broad deployment of distributed applications on clouds, the dominant volume of traffic in cloud networks traverses in an east-west direction, flowing from server to server within a data center. Existing communication solutions are tightly coupled with either the control plane (*e.g.*, pre-programmed model) or the location of compute nodes (*e.g.*, conventional gateway model). The tight coupling makes it challenging to adapt to rapid network expansion, respond to network anomalies (*e.g.*, burst traffic and device failures), and maintain low latency for east-west traffic.

To address this issue, we design Zeta, a scalable and robust east-west communication framework with gateway clusters in large-scale clouds. Zeta abstracts the traffic forwarding capability as a Gateway Cluster Layer, decoupled from the logic of control plane and the location of compute nodes. Specifically, Zeta adopts gateway clusters to support large-scale networks and cope with burst traffic. Moreover, a transparent Multi IPs Migration is proposed to quickly recover the system/devices from unpredictable failures. We implement Zeta based on eXpress Data Path (XDP) and evaluate its scalability and robustness through comprehensive experiments with up to 100k container instances. Our evaluation shows that Zeta reduces the 99% RTT by 5.1× in burst video traffic, and speeds up the gateway recovery by 10.8× compared with the state-of-the-art solutions.

1 Introduction

With an increasing number of distributed applications (*e.g.*, MapReduce [82] and Elasticsearch [32]) on the clouds, east-west communication between instances has become the majority load (even up to 75% [17]) in cloud networks [65]. In addition, cloud providers usually offer isolation for tenants through Virtual Private Cloud (VPC) [77]. Therefore, it is essential for cloud networks to support high-speed and reliable intra-VPC communication [83]. However, two factors

bring much pressure on cloud networks. On one hand, a large-scale cloud can accommodate over 100k servers and millions of instances with Pbps bandwidth [7], bringing congestion risks to the network. According to the monitoring log of a cloud with 1,500 servers, we can observe congestions that last over 1s for more than 12,500 times in one day [38]. On the other hand, containerization leads to centralized startup and short life cycles of instances, which bring great dynamics to the network. For example, Google launches several billion containers per week into Google Cloud [31, 50].

As a result, the east-west communication between instances faces several challenges in large-scale and highly dynamic cloud networks. (1) *Scalability*. The expansion of the instances scale in cloud networks leads to a rapid increase in forwarding rules consumption. For example, the control plane will install 487M rules for a preprogrammed network with 40k instances [22], which brings high latency on the rules lookup and traffic forwarding. Therefore, installment of numerous rules will limit the size of a single VPC and the whole network. (2) *Robustness*. Although the failure probability of a specific equipment is usually low, network abnormal events in large-scale clouds are frequent and inevitable, including device failures [18, 59] and burst traffic [68, 81]. They pose severe network congestion/interruption and degrade the tenants' experience. (3) *Latency*. The latency of configuring forwarding rules and establishing/resuming communication is a crucial metric. When instances launch/migrate, some previous solutions require the control plane to inform all relevant hosts and install/update rules, which especially affects short-lived tasks. For example, a function task (*e.g.*, MilliSort and MilliQuery [47]) usually completes in milliseconds, while it may take a few seconds to launch a function instance and establish connection for it.

The existing east-west communication solutions in cloud networks are usually divided into two main categories. One is the hardware solutions, such as AWS Nitro System [6, 67], Azure FPGA-based SmartNIC [28, 46, 61] and AliCloud P4-based Gateway [57]. The other is the software solutions, including the preprogrammed model (*e.g.*, VMware NSX

*Gongming Zhao and Hongli Xu are the co-corresponding authors.

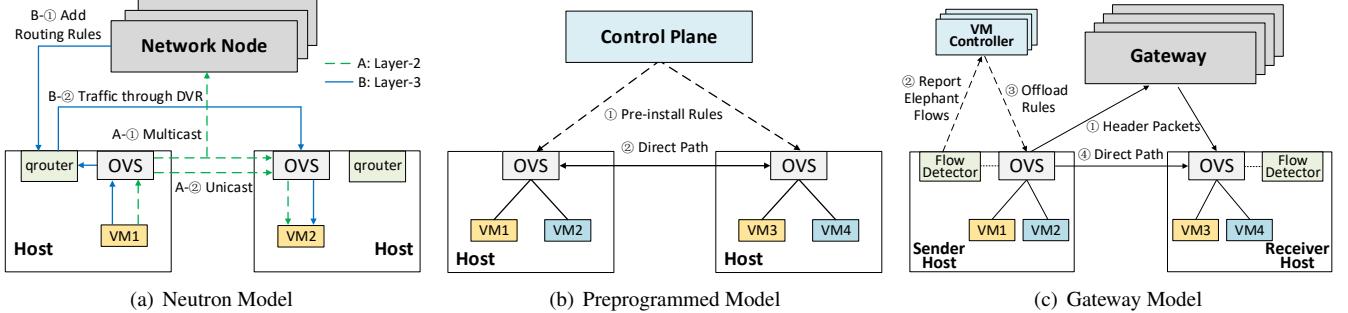


Figure 1: Three Typical East-West Communication Models. (a) Neutron model realizes layer-2 communication by learning MAC address and utilizes DVR (qrouters) for layer-3 communication. (b) Preprogrammed model pre-installs all potential rules when launching VMs. (c) Gateway model pre-installs default rules pointing to the gateway on the host. The gateway forwards the header packets, and the direct path rules of elephant flows will be offloaded to the source hosts.

[39, 56]) and the gateway model (*e.g.*, Google Cloud Hoverboard [22]). Considering the high cost and long development cycles of hardware, software solutions have become the preferred choice for many medium-sized cloud providers. However, the existing software solutions also face several critical disadvantages (see §2.1 for details). First, the preprogrammed model pre-installs numerous rules for VMs and is coupled with the control plane. The conventional gateway model depends on fixed gateways allocated for host zones and is coupled with the location of compute nodes. Hence, they lack the scalability or robustness to adapt to large-scale networks. Second, the existing control loops are complex, which aggravates the recovery delay in network abnormal events, including device failure/overload and VM migration.

To overcome the above challenges, we propose a scalable and robust east-west communication framework in large-scale clouds, called Zeta. Zeta abstracts the traffic forwarding capability as a gateway cluster layer, decoupled from the location and logic of other modules. Specifically, Zeta mainly proposes the following innovative designs. (1) Zeta utilizes gateway clusters to improve the fault tolerance of a single gateway and leverages eXpress Data Path (XDP) [36] to accelerate gateway forwarding, thereby enhancing the network scalability and robustness. (2) Zeta adopts the flow table and group table [84] to realize the intra-cluster gateway load balancing. (3) Zeta proposes Multi IPs Migration to achieve gateway fast recovery, which implements failover by migrating the vIPs of the failed gateways. This scheme avoids updating the on-host default rules pointing to the gateways, making failure recovery transparent to hosts/tenants.

The main contributions of this paper are as follows:

- We analyze the pros and cons of existing typical east-west communication models in large-scale clouds and present the design principles for our framework.
- We design a prototype framework, called Zeta, to achieve scalable and robust east-west communication in large-scale clouds. Zeta is publicly available at <https://github.com/futurewei-cloud/zeta/>.

- We evaluate the robustness and scalability of Zeta through comprehensive experiments. Evaluation results show that Zeta reduces the 99% RTT by 5.1 \times in burst video traffic, and speeds up the gateway recovery by 10.8 \times compared with the state-of-the-art solutions.

2 Background and Motivation

We will analyze the limitations of three typical east-west communication models in large-scale clouds and motivate our work in this section.

2.1 Limitations of Prior Works

As an open source cloud computing architecture, OpenStack helps quickly deploy small-scale clouds [63]. As shown in Figure 1(a), OpenStack Neutron provides the networking capability for the clouds. Specifically, Neutron provides layer-2 networking communication by learning MAC address [55]. When two VMs in the same layer-2 domain communicate for the first time, the source VM will broadcast ARP packets to obtain the MAC address of the destination VM. However, when encountering burst traffic in large-scale networks, it may cause unnecessary layer-2 broadcasts and unicast flooding, leading to poor robustness and scalability [71]. For layer-3 networking, all the traffic will be routed by specific network node(s) in the initial OpenStack releases. It may suffer the risk of network node(s) failure and high forwarding delay in large-scale networks. To this end, OpenStack has released the Distributed Virtual Router (DVR) since Juno version [13], which can significantly mitigate the robustness and latency issues. However, DVR suffers the oversize routing tables and frequent synchronization problems, which also decrease the network scalability [64]. In general, OpenStack gradually improves forwarding performance through evolutions. But due to the lack of targeted designs for large-scale clouds, it still faces robustness and scalability issues.

Table 1: Comparison of the advantages and disadvantages of existing models

Models	Robustness		Scalability		Latency		
	Gateway Failure	Burst Traffic	VPC Size	Global Scale	Forwarding	VM Launching	VM Migration
Neutron [13, 55]	✗	✗	✗	✗	✗	✓	✗
Preprogrammed [39]	—	✓	✗	✗	✓	✗	✗
Gateway [22]	✗	✗	✓	✓	✓	✓	✓
Ours: Zeta	✓	✓	✓	✓	✓	✓	✓

To reduce the forwarding latency between VMs, the preprogrammed model was adopted by many early platforms, such as VMware NSX [39, 56]. As shown in Figure 1(b), the control plane pre-installs all potential rules when launching VMs, as it cannot exactly predict which pairs of VMs will communicate. The traffic between VMs will be forwarded directly with low delays. However, the preprogrammed model brings some nonnegligible system overhead. First, it will pre-install a quadratic number of rules on hosts, which limits the network scalability. Specifically, in a cloud network with h hosts and n VMs, $2n$ rules should be pre-installed before launching a new VM in the worst case, and there will be $O(n \times h)$ rules in the system. A massive number of pre-installed rules will slow down the rules lookup and traffic forwarding, thus limiting the network scale. Second, numerous preprogrammed rules seriously delay the VMs deployment/migration. The control plane needs to pre-install/update all potential rules on hosts, which will cause a significant delay in communication establishment/recovery. For example, the preprogrammed model takes 74 seconds to install 487M rules for a large network with 10k hosts and 40k VMs [22, 39]. Above system overhead leads to poor scalability and flexibility of the preprogrammed model, especially in large-scale cloud networks.

To overcome the disadvantages of the former two models, the gateway model on-demand installs rules, and has been widely adopted by cloud providers, such as Google Cloud Hoverboard [22]. As shown in Figure 1(c), the gateway model organizes all servers into host zones. Host zone/cluster is a collection of colocated machines with uniform network connectivity, each of which is equipped with a master gateway and several backups. This model only pre-installs default rules pointing to the gateway on the host’s vSwitch. When a new flow arrives, the vSwitch sends the header packets to the gateway according to the default rules. Then, the gateway forwards these packets and offloads direct path rules for elephant flows [22], so that the subsequent packets of those elephant flows will be forwarded to the destination directly.

The gateway model improves the network scalability through on-demand rules offloading. However, it allocates a fixed number of gateway(s) to each host zone and may encounter the robustness issues. 1) *Gateway failure*. Although the master-backup gateway model provides disaster tolerance, it will take a long time to migrate all the traffic from the mas-

ter gateway to the backup ones, and cannot effectively cope with gateway failures. For example, it takes 260-310ms [72] to inform 14 affected hosts and update the default entries on each OVS. The recovery delay far exceeds the carrier-grade requirements of 50ms [52, 72, 78]. The network interruption caused by the excessive recovery delay will seriously decrease the QoS. 2) *Burst traffic*. The gateway model only assigns a master gateway to each host zone. When a host zone encounters burst traffic, the corresponding master gateway will be easily overloaded (especially when the control plane cannot detect and offload high bandwidth flows immediately).

2.2 Our Intuitions

As summarized in Table 1, the gateway model combines the advantages of both Neutron and Preprogrammed model in terms of scalability and latency. However, the existing gateway model usually assigns fixed gateway(s) to each host zone. Its gateways incur a high risk of overload/failure under abnormal events, including burst traffic and gateway failures. A natural solution is to deploy multiple master gateways in a host zone to alleviate the impact of burst traffic or abnormal events. However, the gateways need to be provisioned for peak bandwidth usage, making it difficult to efficiently schedule gateway resources.

Another intuitive solution is to organize all gateways into a large virtual cluster to improve disaster tolerance. The new arrival flows will be forwarded to gateways through ECMP [69]. However, once VMs launching/migration occurs, the control plane should notify all gateways to update the forwarding rules, which brings high synchronization overhead on both the gateways and the control plane [60]. For example, assuming that a large datacenter contains 500 gateways and launches 3k containers per second [31, 50]. The controller needs to send 1.5M update messages in one second, which poses a severe risk of control plane overload. Obviously, this solution is not feasible for large-scale clouds.

In order to integrate the pros, but mitigate the cons of models discussed above, we divide all gateways into multiple clusters. A gateway cluster can effectively improve fault tolerance while reducing the synchronization overhead, as the controller only needs to push latest forwarding rules to the gateways of one cluster every time. Moreover, we abstract the gateways’

forwarding capability as Gateway Cluster Layer, which is decoupled from the location and logic of other planes/modules. On the one hand, we utilize gateway clusters independent of the compute nodes to enhance robustness and achieve high performance. We adopt the Multi IPs scheme, which is transparent to hosts/tenants to achieve gateway fast recovery. The independence of gateway cluster gives us flexibility in building high-performance data plane. We choose XDP as the data plane of Zeta, because of its integration with Linux kernel and similar speed as DPDK [24]. On the other hand, the new framework allows easy integration with existing cloud platforms. Thus, we can make full advantage of existing designs, such as Open vSwitch (OVS) [58] group table. According to the above ideas, we design a scalable and robust east-west communication framework in large-scale clouds to support high-performance traffic forwarding.

3 System Design

3.1 Design Goals

Zeta is an east-west communication framework with gateway clusters in large-scale clouds. Our design goals are as follows:

- **Robustness:** High reliability is the core requirement of east-west communication, especially for cloud providers. In particular, Zeta focuses on effectively dealing with burst traffic and abnormal events (*e.g.*, gateway failure/overload/expansion), to avoid network congestion/interruption degrading the tenants' experience.
- **Low Latency:** Since east-west traffic is very sensitive to latency. Zeta aims to reduce the latency of the traffic forwarding through the high-performance in-kernel fast-path. In addition, the lightweight control loop helps reduce the delay of VMs launching/migration.
- **Scalability:** With the rapid growth of cloud scale, Zeta should better support large-scale virtual networks up to 100k instances.
- **Compatibility:** Zeta is open source and can also serve as a common hosting platform to integrate customization network functions into the overall virtual networking.

3.2 System Overview

As shown in Figure 2, to realize the above design goals, we propose an efficient east-west communication framework, called Zeta, which consists of three core modules: Gateway Cluster, On-host Forwarding and Framework Management.

Gateway Cluster Layer establishes a forwarding network based on VXLAN tunnel [48]. It leverages XDP to provide high-performance traffic forwarding and on-demand rules offloading for tenant instances (§4.2). The application

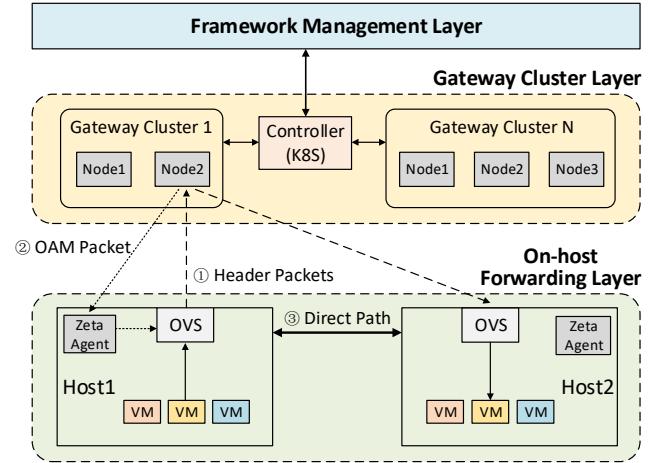


Figure 2: Zeta Framework Overview. Gateway Cluster provides high-performance traffic forwarding and on-demand rules offloading for tenant instances. On-host Forwarding transmits traffic according to default/direct rules and achieves the intra-cluster gateway load balancing through group tables. Framework Management manages the whole network and further improves the system robustness through scheduling.

of gateway cluster ensures better scalability and robustness. Gateways detect the elephant flows and sends OAM (Operations, Administration and Maintenance) packets to the source hosts, which contain direct path rules (§4.3). In addition, Zeta adopts *Multi IPs Migration* to achieve fast recovery from gateway failure/overload/expansion, which makes failure recovery transparent to hosts/tenants (§4.4).

On-host Forwarding Layer transmits traffic according to the rules on OVS. Before deploying a new VPC, a default rule will be pre-installed on the host, which consists of a flow entry and a group entry to achieve the intra-cluster gateway load balancing (§5.1). When two VMs communicate for the first time, the header packets will be sent to a specific gateway according to the default rule. Each host deploys a *Zeta Agent*, which is responsible for parsing OAM packets and installing the direct path rules on the on-host OVS. In addition, the lightweight control loop based on *Zeta Agent* can make a quick response to network adjustments, such as passive instance migration (§5.2).

Framework Management Layer manages the whole network and further enhances the robustness of gateway clusters. When Zeta is initialized, the management layer will determine the VPC-cluster mapping for inter-cluster load balancing (§6.1). To deal with the abnormal events and traffic dynamics, the *Multi IPs Scheduler* will dynamically adjust the configurations (*e.g.*, multi IPs allocation and cluster partition), thereby avoiding overload of partial clusters for better robustness (§6.2).

4 Gateway Cluster Design

4.1 Gateway Cluster Overview

Zeta Gateway Cluster establishes a VXLAN-based forwarding network. Specifically, it provides high-performance traffic forwarding and on-demand rules offloading for tenant instances with scalability and robustness guarantee. As shown in the left plot of Figure 3, Gateway Cluster Layer consists of a cluster controller and several gateway clusters.

Cluster Controller contains management and scheduling logic for gateway clusters. On the one hand, it facilitates the interaction with the Framework Management Layer through its Northbound RESTful API, such as receiving forwarding rules. On the other hand, it manages the gateway clusters and maintains the gateways load balancing through its Southbound API based on gRPC [66]. Cluster Controller is deployed within its own Kubernetes cluster hosted on Zeta control node(s).

Gateway Clusters constitute the data plane of the forwarding network. We divide all gateways into several clusters to achieve the robust gateway forwarding. In practice, each cluster consists of several isomorphic gateways, which store the same forwarding rules to collectively provide traffic forwarding and rules offloading services for tenant instances. Each gateway contains the Forwarding Module (FWD) and the Distributed Flow Table Module (DFT). Specifically, FWD forwards the packets to the destination hosts and offloads direct forwarding rules to the source hosts for those elephant flows. DFT is a lightweight key-value store, which maintains a consistent forwarding table on each gateway of a cluster. When the forwarding table changes (*e.g.*, instances launching/migration), the cluster controller will push the latest rules to each gateway of the corresponding cluster. In addition, there is no state synchronization among gateways (in §4.3).

4.2 XDP-based Traffic Forwarding

The forwarding module of a Zeta gateway is implemented based on XDP [36] to improve the forwarding performance and reduce the transmission latency. XDP is a high-performance and programmable network data path, which can directly process layer-2 frames at the NIC driver and hence bypass the kernel network stack [12, 36, 79]. As illustrated in the right plot of Figure 3, we converge the forwarding, computing and storage functions together, which eliminates the overhead of network stack processing [14, 49].

Forwarding Module works at the NIC driver and can directly operate on raw Ethernet frames. The workflows of XDP-based forwarding program are as follows: (i) Receiving header packets of the source instance from the NIC RX buffer. (ii) Obtaining the forwarding rule of the target instance by querying the storage module, that is, determining the destination host of the traffic. (iii) Parsing the protocol field of VXLAN inner packets. ARP messages will be directly re-

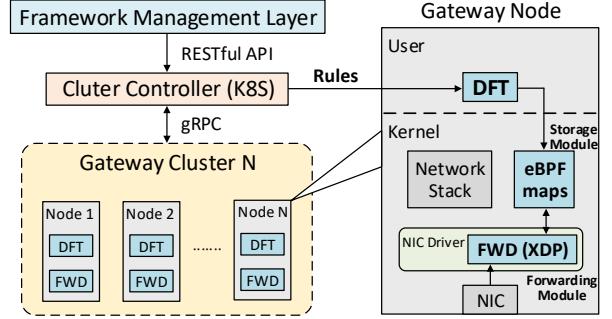


Figure 3: Illustration of Gateway Cluster Design. The left plot is the overview of gateway cluster and the right plot is the implementation details of XDP-based gateway.

sponded to the source instance, while other types of packets will be forwarded to the destination. (iv) Sending OAM (Operations, Administration and Maintenance) packets containing direct rules to the source hosts for the elephant flows.

Storage Module consists of several eBPF maps [2, 19]. These maps are key-value stores [29] that serve as the data channel between DFT and FWD. The forwarding module will also cache the real-time information of flows in eBPF maps. For example, FWD will count the OAM packets generated for each flow to avoid repeatedly offloading one flow.

4.3 Gateway Flow Detection

In order to further reduce the rules stored on the hosts, so as to conserve memory and reduce the forwarding delay caused by rules lookup. Zeta adopts XDP's high-performance packet processing features to detect elephant and mice flows on the gateway, which can improve the efficiency of the detection program and the system's robustness. When encountering burst traffic generated by a simultaneous batch of workloads (*e.g.*, MapReduce [82]), the on-host flow detection program of existing gateway model may be overloaded, as its host agent is usually equipped with limited resources, *e.g.*, 1 CPU core and 1.5GB memory [22]. In contrast, the additional overhead of detecting elephant flows is almost negligible for the XDP-based gateways of Zeta while forwarding traffic.

When traffic arrives at the XDP forwarding module, it will accumulate the total size of each flow in a certain period and store the records in an eBPF LRU Hash map [44, 79]. If the cumulative size of a flow exceeds the threshold (*e.g.*, 20kbps [22]) before the next period, it will be identified as an elephant flow and offloaded to the source hosts. Each flow is only sent to a specific gateway according to the 5-tuple hash (in §5.1), which avoids synchronization of flow size statistics among gateways. In addition, Zeta will monitor the gateway load. When a gateway's CPU or memory utilization reaches the threshold (*e.g.*, 80%), the gateway will pause the elephant flows detection and offload direct rules for all flows.

4.4 Dealing with Failures through Multi IPs

The number of gateways in a cluster will change dynamically due to gateway failures and scaling requirements, and the hash modulo of the default rule will change accordingly (*i.e.*, group entry buckets in §5.1). Therefore, we have to modify all the installed default rules associated with the updated cluster. To this end, massive affected hosts need to be informed, which leads to heavy notification overhead and unacceptable delay [54]. To address this issue, we design the *Multi IPs Migration*. Briefly, each gateway node is logically assigned multiple virtual IPs (vIPs), and the vIPs can be reallocated among nodes. Tenant traffic is bound to vIPs and decoupled from gateways.

The feature of XDP working in the layer-2 networking inspires a solution of gateway failure recovery. We propose the *Multi IPs* scheme to achieve fast failure recovery. Specifically, the cluster controller maintains a *Multi IPs Mapping Table*. When a gateway cluster is initialized, each gateway node in the cluster will be allocated several logical virtual IP-MAC pairs, and send RARP packets [27] to add MAC table entries on the connected ToR switch(es). It should be noted that these vIPs and vMACs are not actually configured in the gateways' NIC, as XDP program can directly operate on the raw Ethernet frames. When a gateway fails, the cluster controller will reassign the logical vIP-vMAC pairs of the failed gateway to other healthy gateways in the cluster. Since the forwarding rules maintained by each gateway in a cluster are consistent, there is no synchronization overhead/delay among gateways during failure recovery. Next, the healthy gateways that have obtained migrated vIP-vMAC pairs will utilize RARP to inform the connected switch(es) to update MAC address table. Then, the packets from instances can be correctly forwarded to healthy gateways.

Figure 4 illustrates an example of fast recovery through *Multi IPs Migration*. Initially, Gateway Cluster 1 contains three gateway nodes, each of which is assigned with two vIP-vMAC pairs, as shown in the *Multi IPs Mapping Table*. When Node2 fails, the cluster controller will update the mapping table, ip3-mac3 and ip4-mac4 originally assigned to the Node2 are reassigned to Node1 and Node3 respectively. Next, Node1 and Node3 utilize the RARP protocol to update the MAC address table of the connected ToR switch, so that the packets toward the failed Node2 will be immediately diverted to the healthy nodes. As a result, the failure recovery is transparent to hosts/tenants without modifying any default OVS entry or on-host ARP cache that involves the failed gateway(s). According to the experiments in §8.3.2, Zeta reduces the average gateway recovery latency from 62ms to 5.5ms.

In conclusion, the *Multi IPs Migration* scheme only needs to update the IPs mapping table and send the RARP packets to ToR switches. The recovery process does not require the participation of control plane or hosts. Therefore, the failure recovery delay and the notification overhead can be almost

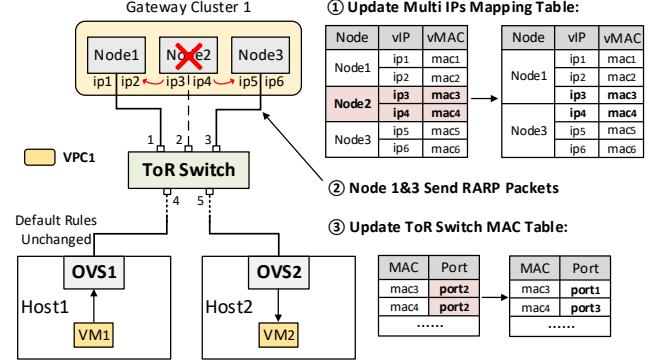


Figure 4: Dealing with Gateway Failures through *Multi IPs*. When Node2 fails, the cluster controller first updates the mapping table to reassign the vIP-vMAC pairs to healthy gateways (*i.e.* Node 1&3). Then Node 1&3 send RARP packets to update the MAC entries on the connected switch. The recovery scheme avoids modifying the default OVS rules on hosts.

negligible. It significantly enhances the robustness of gateway clusters. In addition, the *Multi IPs Migration* can also be applied in (1) Intra-cluster load adjustment and (2) Rapid cluster scaling (covered in §6.2).

5 On-host Forwarding Design

5.1 Load Balancing through Group Tables

This section elaborates on the designs of default entries to achieve intra-cluster gateway load balancing. In order to realize the decoupling of gateway cluster and location (*i.e.*, host or host zone), we construct default rules in VPC granularity. Thus, when launching a new VPC on a compute node, the default rule of this VPC will be pre-installed by Zeta Agent on the on-host OVS.

To achieve the gateway load balancing within a cluster, we utilize the flow table and group table of on-host OVS to orchestrate the gateway clusters. Specifically, each entry of the group table points to a cluster, and the buckets in each group entry specify the gateway nodes in this cluster. When the header packet of a flow reaches OVS, it first matches the flow entry and jumps to a group entry according to the VPC identifier (VPC_id) so that the target cluster for this flow is determined. The VPC-cluster mapping algorithm will be elaborated in §6.1. Then, the packet will be hashed to a bucket in the group entry, which determines the target gateway for this flow. The group entry selects the target gateway based on the 5-tuple hash of a flow. Finally, the load balancing within a gateway cluster can be guaranteed.

We give an example in Figure 5 to illustrate the intra-cluster gateway load balancing with the flow table and group table. Assuming that VM1 belonging to VPC1 communicates with VM3 for the first time. When the header packet arrives at

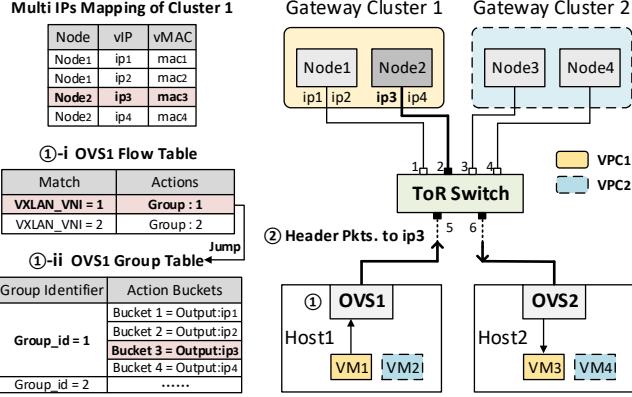


Figure 5: Illustration of Interaction between Flow Table and Group Table. When VM1 belonging to VPC1 communicates with VM3 for the first time, Host1 lookups the OVS1’s default tables, and the default gateway IP of VM1’s flow is ip2. Then, Host1 sends the header packets of VM1 to Node2.

the OVS of Host1, the OVS first matches the flow entry with $VXLAN_VNI=1$ and jumps to the group entry with $Group_id=1$. Each bucket in a group entry corresponds to the IP address of a gateway node in the cluster, and the packet will be hashed to a bucket according to its 5-tuple information. In our example, the packet is hashed to bucket3, that is, the destination address of the packet is ip3. Then, Host1 sends the header packets of VM1 to Node2, and the gateway will forward these packets and offload a direct rule to the source Host1.

5.2 Lightweight Control Agent

The lightweight control loop based on *Zeta Agent* can effectively reduce the recovery latency of the passive instance migration, such as Kubernetes Pod Eviction [42]. In a Kubernetes cluster, when a compute node is out of resources, the Kubernetes scheduler [43] will migrate the relevant pod(s) to other host(s). Conventionally, Kubernetes does not inform its networking plugin (*e.g.*, Flannel [4] and Calico [1]) of pod(s) migration actively. The networking plugin needs to poll Kubernetes database (*e.g.*, Etcd [3]) to obtain the latest pod information. Therefore, the hosts cannot update the installed direct rules immediately. The traffic is still forwarded to the former destination hosts, which results in a network interruption between the affected pods.

Three steps are required in Zeta to restore communication: (i) Obtaining the latest forwarding rules. (ii) Redirecting the packets toward the migrated pods to the correct destination. (iii) Updating the direct rules on the source hosts. We hope *Zeta Agent* remains lightweight to occupy fewer host resources. Meanwhile, Zeta gateways support the above operations. Thus, instead of directly implement above three steps on agent, the traffic towards the migrated pods will be redirected to the gateways and forwarded to the correct destinations.

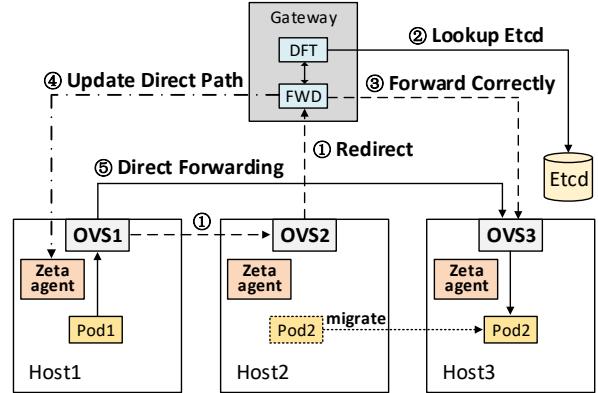


Figure 6: Lightweight Control Agent on compute nodes. When Pod2 is migrated, the flows sent to Pod2 will be redirected to gateway. The gateway forwards the flows and queries the database, then updates the direct path on the source host.

As illustrated in Figure 6, when Pod2 is migrated, the Zeta Agent on Node2 will install an entry on OVS2 to redirect all packets toward the Pod2 to the gateway. FWD on Zeta gateway recognizes the redirected packets and reports their destinations to DFT. DFT queries the latest location information of Pod2 from Kubernetes database and updates the rules cache of FWD. Then, FWD will forward the redirected packets to the correct destination Node3, and send OAM packets to the source Node1. Finally, the Zeta Agent on Node1 will update the direct forwarding rule to Pod2.

6 Framework Management Design

6.1 Gateway Cluster Mapping

When Zeta is initialized, the management layer will determine the VPC-cluster mapping for inter-cluster load balancing.

Gateway Cluster Model. In the Zeta framework, we use $C = \{c_1, c_2, \dots, c_n\}$ to denote the gateway clusters, where $n = |C|$ is the number of clusters. For each gateway cluster c , its forwarding capacity is denoted as $B(c)$. We denote $V = \{v_1, v_2, \dots, v_m\}$ as the VPC set, where $m = |V|$ is the number of VPCs in the cloud. Let $T = \{t_1, t_2, \dots, t_{|T|}\}$ denote the tenants set and each tenant $t \in T$ consists of a VPC set $V_t = \{v_1^t, v_2^t, \dots, v_{|V_t|}^t\}$. Obviously, $V = V_1 \cup V_2 \dots \cup V_{|T|}$. Moreover, the traffic demand of each VPC is denoted as $f(v)$.

Problem Formalization. We define the gateway clusters mapping (GCM) problem in the Zeta framework. To enhance the system robustness and improve the QoS, we need to consider the following two constraints. (1) **VPC Constraint.** A VPC will be mapped to one and only one gateway cluster, as all the vIPs of a group entry belong to the same cluster (§5.1). (2) **Tenant Constraint.** We limit the number of gateway clusters that each tenant can be mapped to. For security reasons,

we do not expect that burst/malicious traffic from a single tenant will affect all gateway clusters.

Moreover, we use binary $x_v^c \in \{0, 1\}$ to denote whether a VPC $v \in V$ is mapped to a gateway cluster $c \in C$ or not. Let binary $y_t^c \in \{0, 1\}$ represent whether the gateway cluster $c \in C$ is assigned the VPCs belonging to tenant $t \in T$ or not. The objective of GCM is to achieve the load-balancing among all gateway clusters. We formulate GCM as follows:

$$\begin{aligned} & \min \lambda \\ \text{s.t. } & \left\{ \begin{array}{ll} \sum_{c \in C} x_v^c = 1, & \forall v \in V \\ \sum_{v \in V} x_v^c \cdot f(v) \leq \lambda B(c), & \forall c \in C \\ x_v^c \leq y_t^c, & \forall v \in V, t \in T, c \in C \\ \sum_{c \in C} y_t^c \leq k, & \forall t \in T \\ x_v^c \in \{0, 1\}, & \forall v \in V, c \in C \\ y_t^c \in \{0, 1\}, & \forall t \in T, c \in C \end{array} \right. \quad (1) \end{aligned}$$

The first set of equations means that all traffic of a VPC will be forwarded to one gateway cluster by default. The second set of inequalities describes the traffic load on each gateway cluster, where $\lambda \in [0, 1]$ represents the load balancing factor. The third set of inequalities indicates that the tenant t is mapped to gateway cluster c only if VPC(s) of tenant t is processed by cluster c . The fourth set of inequalities represents the *Tenant Constraint*, that is, the VPCs of a tenant will be mapped to at most k gateway clusters. Our objective is to achieve the load balancing among all gateway clusters, *i.e.*, minimizing the load balancing factor λ .

We give an empirical formula to set the tenant constraint k in §A.1, and propose a rounding-based algorithm for the VPC-cluster mapping in §A.2.

6.2 Multi IPs Scheduler

The *Multi IPs Scheduler* executes the IPs migration scheme proposed in §4.4. It dynamically updates the IPs allocations to eliminate the overload of gateway clusters caused by the burst traffic and abnormal events. In practice, when a gateway exceeds the load threshold (*e.g.*, 80%), it will immediately report such overload to the control plane. Then the *Multi IPs Scheduler* starts to perform the following two steps:

Step 1: Intra-Cluster Load Adjustment. The scheduler first sorts all gateways of a cluster in the descending order of their load. Next, the scheduler attempts to migrate a vIP-vMAC pair from the overloaded gateway to the gateway with the lightest load, and re-sorts gateways' load. Then, the scheduler will repeat above IPs migration and gateway sorting procedure until none of the gateways in the cluster is overloaded. If we cannot eliminate the overloaded gateways with step 1, the scheduler will go to step 2.

Step 2: Cluster Scaling. If a cluster cannot eliminate overload through internal load adjustment, *e.g.*, a legitimate VPC

has burst traffic. The scheduler will migrate gateways from other clusters to this cluster or expand new gateways for this cluster. The scheduler first sorts all the clusters by their average load in the descending order and attempts to reassign a gateway from the least loaded cluster to the overloaded cluster. We can utilize *Multi IPs Migration* to achieve rapid gateway migration among clusters. However, if the gateway migration causes overload risk to the source cluster, the scheduler will directly expand the overloaded cluster with a new gateway.

7 Implementation

We implement Zeta based on Linux 5.4 kernel. The Cluster Controller includes 3k lines of Python code, the XDP-based gateway forwarding function includes 4.5k lines of C code, and the Zeta Agent includes 2k lines of C++ code.

Zeta provides two deployment methods. One is based on physical machines, and we give a best practice in §B.2. The other is based on Kernel-based VMs (KVMs), which can quickly deploy dozens of KVM-based gateways on several physical machines (see §B.3 for more details).

8 Evaluation

We first conduct an ablation analysis to measure the performance of Zeta gateway. We then test the robustness of Zeta under burst traffic and abnormal events. Finally, we evaluate the scalability of Zeta in public and private cloud scenarios.

8.1 Experimental Setting

Testbed Setups. We use 23 servers to build the testbed, all running Ubuntu 18.04 with Linux kernel 5.4. Considering our limited number of servers, we deploy KVM-based gateways on several physical machines to simulate gateway clusters. In addition, we launch a large number of container instances on each compute node to evaluate scalability, because of limited number of compute nodes. The scalability in this paper mainly refers to the instance scale, instead of the host scale, as the forwarding rules stored in the gateways and the tenant traffic depend on the instance scale.

Specifically, 20 servers are compute nodes, each equipped with dual 22-core Intel Xeon 6161 CPUs, 640GB memory and an Intel XL710 40GbE NIC. The other 3 servers are used to deploy gateway clusters, each equipped with dual 16-core Intel Xeon E5-2697A CPUs, 256GB memory and an Intel XL710 40GbE NIC. We deploy a total of 45 KVM-based gateways on the 3 physical gateway machines. Each KVM-based gateway is equipped with 4 vCPUs and 16GB memory. For Zeta, we divide the 45 gateways into 10 clusters.

Moreover, according to the empirical data in [22], we set the rules offloading threshold to 20kbps on the gateway.

Benchmarks. We compare the robustness and scalability of Zeta with other three typical frameworks. The first framework is the conventional gateway model [22], called GWZone, and its gateway is modified based on the implementations of Zeta’s gateway. It allocates a master gateway for each host zone and equips backups to deal with gateway failure. Unlike Zeta, GWZone detects elephant flows on compute nodes. When GWZone faces gateway failure, it will update the default entries on affected hosts and migrate traffic to the backup gateways. We equip GWZone with 9 additional backup KVM-based gateways. As the backup gateways only consume ~ 0.1 vCPU and $\sim 2\text{GB}$ memory in standby, they will not affect the performance of the master gateways. The second one is the OpenStack Neutron [55], which provides layer-2 networking communication by learning MAC address. The third one is the Preprogrammed model, which is a simplified implementation of VMWare NSX [39, 56] as it is not open source. The Preprogrammed model will pre-install all potential rules before launching VMs.

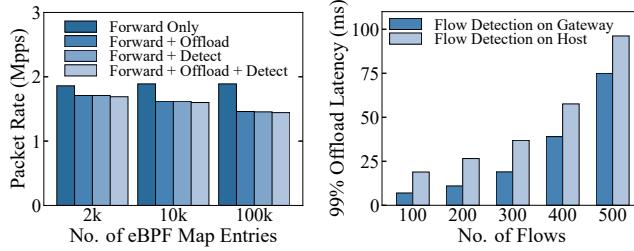


Figure 7: Packet Rate of a Physical Core vs. Entries

Figure 8: 99% Offloading Latency vs. No. of Flows

8.2 Microbenchmark

We first evaluate the impact of flow detection and rules offloading on forwarding performance with a physical core. We use iPerf [37] to generate UDP traffic, and the inner packet size is 64 bytes. In addition, the number of entries stored in eBPF map ranges from 2k to 100k. As shown in Figure 7, a single physical core can forward 1.86M packets per second under 2k entries. When the rule offloading or flow detection is supplied, the forwarding rate reduces by 8.1% to 1.71Mpps, as these two functions introduce additional eBPF map read/writes for flow statistics. After adding both flow detection and offloading functions on the gateway, the performance decreases slightly. For example, the forwarding rate only reduces by 1.2% from 1.71Mpps to 1.69Mpps under 2k entries. This is because the map read/writes are the majority overhead for forwarding, while the detection and offloading functions require the same number of map read/writes. When the number of entries scales to 100k, the forwarding rate with rule offloading and flow detection drops by 14% to 1.45Mpps, as the timeout mechanism of maps for flow statistics leads to throughput degradation with the number of entries increasing. We will optimize the timeout mechanism in future work.

We then measure the rules offloading latency with flow detection on gateway and host. The gateway still performs traffic forwarding and rules offloading with a physical core. We use iPerf to generate UDP flows on a host, each of which is 10Mbps. Figure 8 shows that flow detection on gateway can reduce the 99th percentile of offloading latency by 22% under 500 flows compared with that on host, as the performance of on-host detection is worse than XDP on gateways.

In general, flow detection on gateway can reduce the rules offloading latency (*e.g.*, reduce 22% as shown in Figure 8) and has little impact on the forwarding performance (*e.g.*, decrease 1.2% from 1.71Mpps to 1.69Mpps as shown in Figure 7). Thus, Zeta detects elephant flow on gateways for faster rules offloading with little detection overhead.

We also evaluate the linear scaling throughput of Zeta gateways (§C.2).

8.3 Robustness Evaluation

In this section, we evaluate the performance of Zeta under various burst traffic workloads and different abnormal events. Based on the further transformation (§C.1) of Google cluster-data [30], we deploy 100 VPCs with 2,000 VMs on the 20 compute nodes. Each VPC contains 10-90 VMs, and each VM is equipped with 1 vCPU and 6GB memory.

8.3.1 Robustness under Burst Traffic

We compare the robustness of the Zeta gateway cluster with GWZone under burst traffic of different applications. We choose three typical traffic workloads according to the traffic characteristics in cloud networks [8, 45], including MapReduce, video and audio. Specifically, we deploy a MapReduce cluster in each VPC and execute the word-counting application on each MapReduce cluster simultaneously with input size of 10GB, which mainly generates TCP elephant flows. We also deploy video and audio applications in each VPC. The video traffic contains UDP elephant flows with bandwidth ranging from 2.4Mbps (720P video) to 100Mbps (8K video) [10, 15]. The audio traffic consists of UDP mice flows whose transmission rate ranges from 12.2kbps to 23.85kbps [41].

Figures 9-12 illustrate the performance metrics of Zeta gateways under different burst traffic scenarios. Zeta assigns a gateway cluster to each VPC, while GWZone assigns a master gateway to each host zone. Thus, Zeta can achieve better load balance to deal with various burst traffic. For example, Figure 9 shows that Zeta can reduce the maximum gateway load by 18.5%, 33.9% and 25.2% compared with GWZone in the three applications, respectively. In addition, it is noteworthy that the acknowledgment and retransmission mechanism of MapReduce’s TCP flows further increase the gateway load, which leads to the highest gateway load compared with video and audio streams. Moreover, Figure 11 shows the 99th percentile of normalized FCT, which is normalized to the FCT

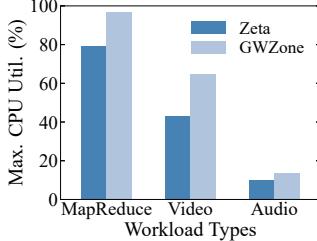


Figure 9: Max. Gateway CPU Utilization vs. Workload Types

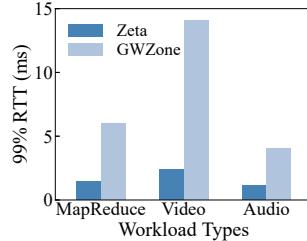


Figure 10: 99% RTT vs. Workload Types

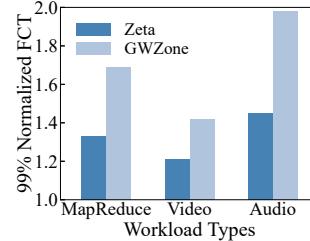


Figure 11: 99% Normalized FCT vs. Workload Types

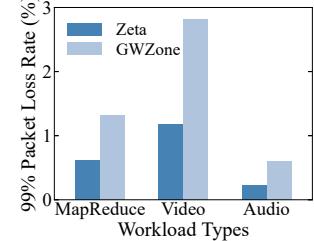


Figure 12: 99% Packet Loss Rate vs. Workload Types

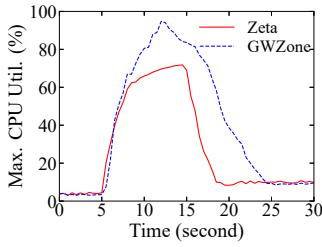


Figure 13: Max. Gateway CPU Utilization over Time

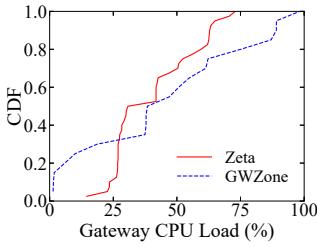


Figure 14: CDF of Gateway CPU Utilization

without burst traffic. The 99% normalized FCT achieved by Zeta is 21.3%, 14.8% and 26.8% lower than that of GWZone under three scenarios, respectively. Although the gateway load of audio traffic is low, it mainly consists of mice flows, which will be forwarded by gateways without offloading direct path rules. Thus, the cumulative delay of the audio flows caused by gateway forwarding will be the largest among the three applications, which results in the maximum FCT of audio flows. Besides, we observe from Figure 12 that the 99th percentile of packet loss rate of Zeta under the three scenarios reduces by 53.8%, 58.2% and 63.3% compared with GWZone. The above results prove that Zeta can effectively conquer different burst traffic and avoid gateways overhead.

Furthermore, we evaluate several performance metrics of Zeta in burst video traffic compared with other frameworks, as shown in Figures 13-18. During an interval of 0.5s, we record the CPU utilization of gateways, number of offloaded rules, rule offloading latency and FCT. Specifically, Figure 13 shows the maximum gateway load of Zeta and GWZone in 10k burst video flows. According to the experimental settings, burst traffic are generated randomly in 5-15s, so the gateway load increases sharply at the 5th second. Next, Zeta detects elephant flows faster on the gateways, so it quickly achieves the balance between offloading and newly coming elephant flows. However, the mice flows continue to increase, so the load of Zeta between 7-15s increases slightly on the basis of stability. Meanwhile, the on-host flow detection of GWZone suffers from high latency, and the elephant flows can not be offloaded in time. Thus, the loads of GWZone’s gateways increase sharply from 5s to 13s.

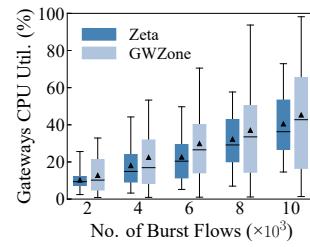


Figure 15: Gateway CPU Utilization vs. No. of Burst Flows

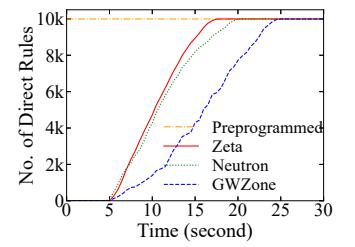


Figure 16: No. of Offloaded Direct Rules over Time

To further study how the workload of gateways distributes, we show the gateways’ CPU utilization at the 12th second, when Zeta and GWZone both suffer high gateway workload, in Figure 15. Zeta achieves lower average load with more concentrated load distribution than GWZone, which means better load balancing. Figure 14 shows the load CDF of gateways in 10k burst video flows. GWZone’s backup gateways are lightly loaded, while 25% master gateways are overloaded (*i.e.*, the CPU load exceeds 80%). The above results show the superiority of Zeta gateway cluster in load balancing.

Figure 16 shows the number of offloaded direct forwarding rules in 10k burst video flows. Due to the latency of the on-host flow detection program, the number of offloaded rules for GWZone increases slowly. The number of Zeta offloading rules is increasing rapidly. Preprogrammed is constant at a high point as its preprogrammed model. The trend of Neutron is similar to Zeta. Figure 18 shows CDF of Normalized FCT in 10k burst video flows. The results are similar to Figure 16. The preprogrammed model performs the best, followed by Zeta and Neutron, while GWZone is the worst.

8.3.2 Fast Recovery from Abnormal Events

We measure the recovery latency of Zeta under abnormal events. Zeta adopts *Multi IPs Migration* for fast recovery, while GWZone updates the default OVS entries on hosts.

Considering that anomaly detection is usually performed by polling, we hope that the delay measurement of failure recovery can avoid the error caused by polling interval. Specifically, we first sequentially send Ping probe every 0.5ms. We

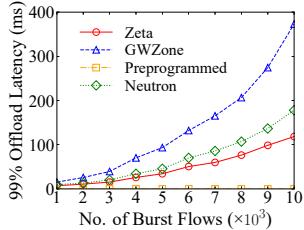


Figure 17: 99% Offload Latency vs. No. of Burst Flows

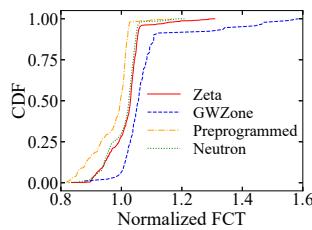


Figure 18: CDF of Normalized FCT

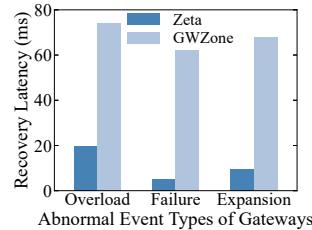


Figure 19: Recovery Latency vs. Abnormal Events

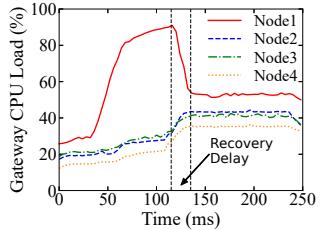
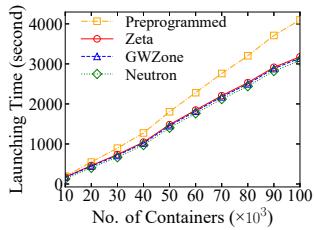
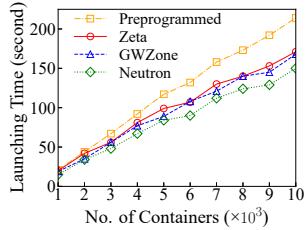


Figure 20: CPU Load of Gateways in a Cluster over Time



(a) The public cloud



(b) The private cloud

Figure 21: Launching Time vs. No. of Containers

make an artificial abnormal event and notify the controller immediately. Then, the controller performs the IPs Migration. By counting the number of lost packets during the failure recovery, we derive the recovery delay.

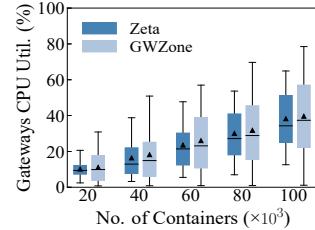
From the results in Figure 19, we observe that Zeta can greatly reduce the recovery latency of the three abnormal events compared with GWZone. For example, the gateway failure recovery delay of Zeta is 5.5ms, which is $\sim 10.8 \times$ faster than that of GWZone, because GWZone needs to inform each host and update ~ 100 default entries on each OVS.

Figure 20 illustrates the load status of each gateway in a cluster of Zeta during the overload event. Specifically, the burst flows with default destination of Node1 arrive in 35ms, and the CPU load of Node1 increases rapidly. When the gateway's CPU utilization reaches the 90% threshold, the *Multi IPs Migration* is triggered in 120ms, and three vIPS on Node1 are reassigned to the other three nodes with lighter load. Then, the load of Node1 quickly decreases to a normal level within 19.5ms. It is intuitive that *Multi IPs* can effectively conquer the overload of a single gateway and rapidly adjust the load imbalance of intra-cluster.

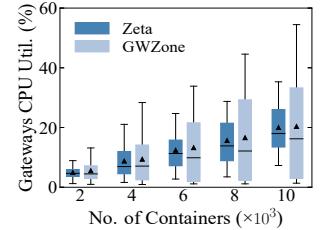
8.4 Scalability Evaluation

In this section, we evaluate the scalability of Zeta in both public and private cloud scenarios. We first measure the latency of launching up to 100k container instances. Then, we evaluate the performance metrics of Zeta and GWZone under the large-scale cloud network.

The public cloud scenario contains a large number of instances/VPCs. Based on the transformation (§C.1) of Google



(a) The public cloud



(b) The private cloud

Figure 22: Gateway CPU Utilization vs. No. of Containers

cluster-data [30], we deploy 568 tenants and 1885 VPCs with up to 100k containers on the 20 compute nodes. Each VPC contains 2-364 containers. The private cloud scenarios have a small number of VPCs/tenants, but a VPC may contain a large number of instances. We deploy 52 tenants and 90 VPCs with up to 10k containers on the 20 compute nodes, and each VPC has a number of instances ranging from 2 to 2765.

According to the bandwidth distribution of flows in [22], we let 16% of container pairs communicate, and the traffic intensity of each flow ranges from 10kbps to 1Gbps.

8.4.1 Large-Scale Instances Launching

Figure 21 shows that the on-demand rules offloading model has a lower instance deployment latency compared with the preprogrammed model when spawning a large number of instances in a large-scale cloud network. For example, when launching 100k containers in the public cloud environment, Zeta spends 3178 seconds and installs 12k default forwarding rules, while Preprogrammed spends 4097 seconds and programs a total of 3.4M rules. That is, Zeta reduces the launching time by 24% and the number of rules by $278 \times$ compared with Preprogrammed. The reason for the above results is that the on-demand rules offloading can avoid pre-installing numerous entries for instances that never communicate with each other, thus it reduces the latency of instances launching.

8.4.2 Large-Scale Instances Communication

Figures 22-24 show the advantages of Zeta gateway cluster under large-scale networks. As shown in Figure 22, the average

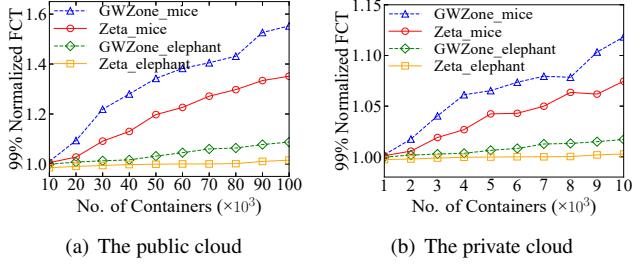


Figure 23: 99% Normalized FCT vs. No. of Containers

load of Zeta gateways is close to that of GWZone. However, Zeta gateways achieve more concentrated load distribution than GWZone and there is a big gap between maximum and minimum load of GWZone gateways, which means the superiority of Zeta gateway cluster in load balancing.

Next, we evaluate the impact of Zeta and GWZone gateways on FCT. The Normalized FCT of elephant flows and mice flows are calculated respectively. Figure 23 shows that though Zeta and GWZone have the similar normalized FCT, Zeta still outperforms GWZone by 7% in public cloud scenario, as there is no flow detection load on hosts. In addition, the FCT of elephant flows are both smaller than that of mice flows, because the elephant flows will be forwarded directly.

Finally, we evaluate the packet loss rate of Zeta and GWZone with offloaded elephant flows and non offloaded mice flows to prove the scalability of Zeta. Figure 24 shows that the packet loss rate of Zeta is lower than that of GWZone because of the better load balancing effect of Zeta gateway cluster. For example, in public cloud with the network scale of 100k containers, the packet loss rate of elephant flows and mice flows of Zeta is 24% and 37% lower than that of GWZone, respectively. In addition, the packet loss rate of elephant flows is higher than that of mice flows. The reason is that these elephant flows will be forwarded by the gateways at the beginning, and burst traffic will cause the gateways overload, resulting in a higher packet loss rate. Therefore, the packet loss of elephant flows is mainly concentrated in the initial gateway forwarding period, and the packet loss of direct path forwarding after offloading will be significantly reduced.

9 Related Work

Cloud and datacenter virtual networks. There are a multitude of researches on the cloud/datacenter virtual networks, including control plane [21, 26, 35, 40, 73] and data plane [22, 39, 55, 61]. As a crucial solution, overlay network adopts tunnel encapsulation protocols (*e.g.*, VXLAN [48], NVGRE [70], Geneve [33], etc) to build the scalable and flexible virtual networks. Virtual network devices (*e.g.*, vSwitch [58, 76], vRouter [69] and gateway [22, 57]) are essential in the cloud networks, as they are dedicated to provide efficient, secure and stable connections for tenants in clouds. In this paper, we

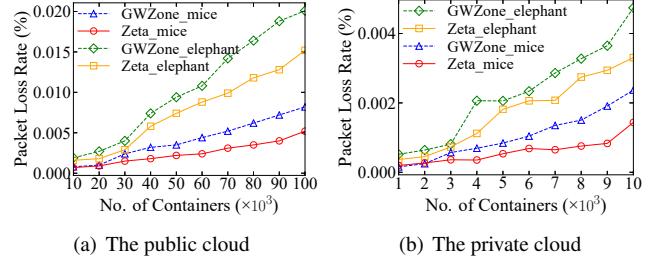


Figure 24: Packet Loss Rate vs. No. of Containers

focus on improving the robustness of east-west forwarding with the designs of gateway cluster and multi IPs migration.

High performance and programmable data plane. Data plane is the most performance-critical part of the cloud networks, which is usually accelerated with specialized hardware components and sophisticated software methods [9]. In hardware, ASIC [57, 75], FPGA [16, 28, 46, 61] and network processor [51, 53] can provide high-throughput and low-latency packet processing. In contrast, software methods have the advantage of fast and flexible iteration, including DPDK [24], XDP [36], Netmap [62], etc. Though XDP is not the first mover in this area, we choose XDP as the data plane of Zeta, because of its integration with Linux kernel, interaction with other kernel components and similar speed as DPDK.

eBPF and its applications. eBPF is an instruction set and an execution environment inside the Linux kernel [79]. It enables injecting custom code into the kernel through the hooks. eBPF is extensively used in security [25], tracing [11] and networking [20]. XDP is one of the most widely used eBPF hooks for high-performance packet processing that can bypass the kernel network stack [36].

10 Conclusion and Future Work

In this paper, we propose a scalable and robust east-west communication framework in large-scale clouds, called Zeta. Comprehensive experiment results show high robustness and scalability of Zeta. For example, Zeta speeds up the gateway failure recovery by 10.8 \times compared with the existing solutions. In future, we will optimize the timeout mechanism of eBPF map to reduce the impact on forwarding performance.

Acknowledgments

We thank our shepherd Minlan Yu and anonymous reviewers for their insightful comments. We also thank the open-source community of Zeta project founded by Futurewei Technologies in 2019 and paper benefits from the original design and implementation of Zeta project. The authors from USTC are supported in part by the National Science Foundation of China under Grant 62102392 and the National Science Foundation of Jiangsu Province under Grant BK20210121.

References

- [1] Calico Project, 2022. <https://www.tigera.io/project-calico/>.
- [2] eBPF Maps, 2022. <https://ebpf.io/what-is-ebpf/#maps>.
- [3] Etcd: A distributed, reliable key-value store, 2022. <https://etcd.io/>.
- [4] Flannel Project, 2022. <https://github.com/flannel-io/flannel>.
- [5] David Ahern. XDP and the cloud: Using XDP on hosts and VMs, 2020. <https://legacy.netdevconf.info/0x14/pub/slides/24/netdev-0x14-XDP-and-the-cloud.pdf>.
- [6] Amazon AWS. AWS Nitro System, 2022. <https://aws.amazon.com/ec2/nitro/>.
- [7] Victor Bahl. Emergence of micro datacenter (cloudlets/edges) for mobile computing. *Microsoft Devices & Networking Summit 2015*, 5, 2015.
- [8] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.
- [9] Roberto Bifulco and Gábor Rétvári. A survey on the programmable data plane: Abstractions, architectures, and open problems. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–7. IEEE, 2018.
- [10] Kashif Bilal and Aiman Erbad. Impact of multiple video representations in live streaming: A cost, bandwidth, and QoE analysis. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 88–94. IEEE, 2017.
- [11] bpftrace. High-level tracing language for Linux systems, 2022. <https://bpftace.org/>.
- [12] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990, 2020.
- [13] Tuan Anh Bui and Marco Canini. *Cloud network performance analysis: an OpenStack case study*. PhD thesis, Master’s thesis, Université Catholique de Louvain, 2016.
- [14] Qizhe Cai, Shubham Chaudhary, Midhul Vuppala, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 65–77, 2021.
- [15] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G Andersen, Michael Kaminsky, and Subramanya R Dulloor. Scaling video analytics on constrained edge nodes. *arXiv preprint arXiv:1905.13536*, 2019.
- [16] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.
- [17] Qixiang Cheng, Meisam Bahadori, Madeleine Glick, Sébastien Rumley, and Keren Bergman. Recent advances in optical technologies for data centers: a review. *Optica*, 5(11):1354–1370, 2018.
- [18] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 342–356, 2018.
- [19] Cilium. BPF and XDP Reference Guide, 2022. <https://docs.cilium.io/en/latest/bpf/>.
- [20] Cilium. eBPF-based Networking, Security, and Observability, 2022. <https://cilium.io/>.
- [21] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 254–265, 2011.
- [22] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermenio, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, 2018.
- [23] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with SR-IOV. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

- [24] DPDK. Data Plane Development Kit, 2022. <https://www.dpdk.org/>.
- [25] Falco. Cloud Native Runtime Security, 2022. <https://falco.org/>.
- [26] Andrew D Ferguson, Steve 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.
- [27] Ross Finlayson, Timothy Mann, JC Mogul, and Marvin Theimer. RFC0903: Reverse Address Resolution Protocol, 1984.
- [28] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.
- [29] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *NSDI*, pages 487–501, 2021.
- [30] Google. Google cluster-data, 2020. <https://github.com/google/cluster-data>.
- [31] Google Cloud. Containers at Google, 2022. <https://cloud.google.com/containers>.
- [32] Clinton Gormley and Zachary Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. "O'Reilly Media, Inc.", 2015.
- [33] Jesse Gross, T Sridhar, P Garg, C Wright, I Ganga, P Agarwal, K Duda, D Dutt, and J Hudson. Geneve: Generic network virtualization encapsulation. *IETF draft*, 2014.
- [34] Gurobi. The Fastest Solver, 2022. <https://www.gurobi.com/>.
- [35] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 19–24, 2012.
- [36] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.
- [37] iPerf. The TCP, UDP and SCTP network bandwidth measurement tool, 2022. <https://iperf.fr/>.
- [38] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 202–208, 2009.
- [39] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, et al. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, 2014.
- [40] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.
- [41] Linda Kozma-Spytek, Paula Tucker, and Christian Vogler. Voice telephony for individuals with hearing loss: The effects of audio bandwidth, bit rate and packet loss. In *The 21st International ACM SIGACCESS Conference on Computers and Accessibility*, pages 3–15, 2019.
- [42] kubernetes Project. kubernetes Eviction Policy, 2022. <https://kubernetes.io/docs/concepts/scheduling-eviction/eviction-policy/>.
- [43] kubernetes Project. Kubernetes Scheduler, 2022. <https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/>.
- [44] Martin KaFai Lau. bpf: Improve LRU map lookup performance, 2017. <https://patchwork.ozlabs.org/project/netdev/cover/20170901062713.1842249-1-kafai@fb.com/>.
- [45] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-driven bandwidth guarantees in datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 467–478, 2014.
- [46] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14, 2016.

- [47] Yilong Li, Seo Jin Park, and John K Ousterhout. Millisort and milliquery: Large-scale data-intensive computing in milliseconds. In *NSDI*, pages 593–611, 2021.
- [48] Mallik Mahalingam, Dinesh G Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreger, T Sridhar, Mike Bursell, and Chris Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. *RFC*, 7348:1–22, 2014.
- [49] Ilias Marinos, Robert NM Watson, and Mark Handley. Network stack specialization for performance. *ACM SIGCOMM Computer Communication Review*, 44(4):175–186, 2014.
- [50] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. Bastion: A security enforcement network stack for container networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 81–95, 2020.
- [51] Netronome. Agilio CX SmartNICs, 2022. <https://www.netronome.com/products/agilio-cx/>.
- [52] B Niven-Jenkins, D Brungard, M Betts, N Sprecher, and S Ueno. Requirements of an MPLS transport profile, 2009.
- [53] Nvidia/Mellanox. Nvidia BlueField Data Processing Units, 2022. <https://www.nvidia.com/en-us/networking/products/data-processing-unit/>.
- [54] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 125–139, 2018.
- [55] OpenStack Project. OpenStack Basic Networking, 2022. <https://docs.openstack.org/neutron/latest/admin/intro-basic-networking.html>.
- [56] Marcus Oppitz and Peter Tomsu. Software defined virtual networks. In *Inventing the Cloud Century*, pages 149–200. Springer, 2018.
- [57] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 194–206, 2021.
- [58] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.
- [59] Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 9–22, 2013.
- [60] Konstantinos Poularakis, Qiaofeng Qin, Liang Ma, Sastry Kompella, Kin K Leung, and Leandros Tassiulas. Learning the optimal synchronization rates in distributed SDN control architectures. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1099–1107. IEEE, 2019.
- [61] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.
- [62] Luigi Rizzo. Netmap: A novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.
- [63] Tiago Rosado and Jorge Bernardino. An overview of openstack architecture. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, pages 366–367, 2014.
- [64] Arsalan Saghir and Tahir Masood. Performance evaluation of openstack networking technologies. In *2019 International Conference on Engineering and Emerging Technologies (ICEET)*, pages 1–6. IEEE, 2019.
- [65] Ken-ichi Sato, Hiroshi Hasegawa, Tomonobu Niwa, and Toshio Watanabe. A large-scale wavelength routing optical switch for data center networks. *IEEE Communications Magazine*, 51(9):46–52, 2013.
- [66] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A remote procedure call API for grid computing. In *International Workshop on Grid Computing*, pages 274–278. Springer, 2002.
- [67] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 40(6):67–73, 2020.
- [68] Danfeng Shan, Fengyuan Ren, Peng Cheng, Ran Shu, and Chuanxiong Guo. Observing and mitigating microburst traffic in data center networks. *IEEE/ACM Transactions on Networking*, 28(1):98–111, 2019.

- [69] Hua Shao, Xiaoliang Wang, Yuanwei Lu, Yanbo Yu, Shengli Zheng, and Youjian Zhao. Accessing cloud with disaggregated software-defined router. In *NSDI*, pages 1–14, 2021.
- [70] Murari Sridharan. Nvgre: Network virtualization using generic routing encapsulation. *draft-sridharan-virtualization-nvgre-00.txt*, 2011.
- [71] Piyush Raman Srivastava and Saket Saurav. Networking agent for overlay L2 routing and overlay to underlay external networks L3 routing using OpenFlow and Open vSwitch. In *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 291–296. IEEE, 2015.
- [72] Dimitri Staessens, Sachin Sharma, Didier Colle, Mario Pickavet, and Piet Demeester. Software defined networking: Meeting carrier grade requirements. In *2011 18th IEEE workshop on local & metropolitan area networks (LANMAN)*, pages 1–6. IEEE, 2011.
- [73] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 426–439, 2016.
- [74] The kernel development community. Linux TUN/TAP Device, 2022. <https://www.kernel.org/doc/html/latest/networking/tuntap.html>.
- [75] Barefoot Tofino. World’s fastest P4-programmable Ethernet switch ASICs, 2018.
- [76] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open vSwitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 245–257, 2021.
- [77] Venkatanathan Varadarajan, Yingqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 913–928, 2015.
- [78] Luis Velasco. Recovery mechanisms in ASON/GMPLS networks. *Universitat Politècnica de Catalunya (UPC), Barcelona, Spain*, 2009.
- [79] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.
- [80] Jason Wang. Accelerating VM networking through XDP, 2017. https://events19.linuxfoundation.cn/wp-content/uploads/2017/11/Accelerating-VM-Networking-through-XDP_Jason-Wang.pdf.
- [81] Jingzhou Wang, Gongming Zhao, Hongli Xu, Yutong Zhai, Qianyu Zhang, He Huang, and Yongqiang Yang. A robust service mapping scheme for multi-tenant clouds. *IEEE/ACM Transactions on Networking*, 2021.
- [82] Weina Wang, Kai Zhu, Lei Ying, Jian Tan, and Li Zhang. Maptask scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality. *IEEE/ACM Transactions On Networking*, 24(1):190–203, 2014.
- [83] Lizhao You, Hao Tang, Jiahua Zhang, and Xiao Li. Fast configuration change impact analysis for network overlay data center networks. In *4th Asia-Pacific Workshop on Networking*, pages 8–15, 2020.
- [84] Gongming Zhao, Hongli Xu, Shigang Chen, Liusheng Huang, and Pengzhan Wang. Joint optimization of flow table and group table for default paths in sdns. *IEEE/ACM Transactions on Networking*, 26(4):1837–1850, 2018.

A Additional Details of Cluster Mapping

A.1 Empirical Formula for Tenant Constraint

We use $C = \{c_1, c_2, \dots, c_n\}$ to denote the gateway clusters, where $n = |C|$ is the number of clusters. In addition, let I_t denote the number of instance owned by tenant $t \in T$. Then, we use the following empirical formula to set the tenant constraint k :

$$k = \lceil \frac{\max_{t \in T} \{I_t\}}{\sum_{t \in T} I_t} \times n \rceil + 1 \quad (2)$$

For example, when our testbed in §8.4 contains 100k container instances, the largest tenant has 27652 instances. We set the tenant constraint $k = 4$, which means the VPCs of a tenant will be mapped to at most 4 gateway clusters.

A.2 Rounding-Based Algorithm

To solve the problem in Eq. (1), we propose a rounding-based gateway cluster mapping (RGCM) algorithm for the GCM problem. The RGCM algorithm includes two steps. The first step is to construct a relaxed version of GCM, named LP-GCM, by relaxing the variable binary constraints. Specifically, LP-GCM assumes that each flow can be splittable and forwarded to multiple gateway clusters. Since LP-GCM is a linear programming, we can derive the fractional solutions $\{\tilde{x}_v^c\}$ and $\{\tilde{x}_t^c\}$ with an optimization solver, such as Gurobi [34]. The optimal fractional result is denoted as $\tilde{\lambda}$.

The second step is to derive the integer solutions with rounding scheme. The integer solutions are denoted as $\{\tilde{x}_v^c\}$

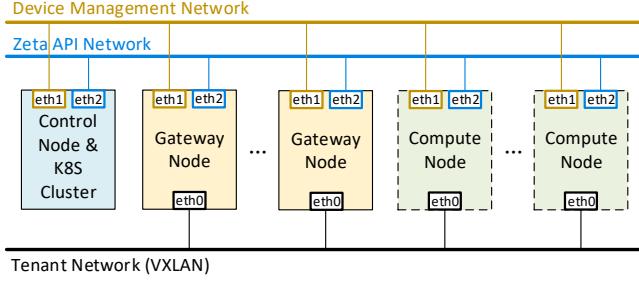


Figure 25: Best Practice for Zeta Physical Deployment.

and $\{\hat{y}_t^c\}$. For each tenant $t \in T$, RGCM first sorts each gateway cluster $c \in C$ by the value of \hat{y}_t^c in the descending order. Then RGCM sets the top k maximum \hat{y}_t^c to 1, which means that the traffic of tenant t can be processed by these k gateway clusters. The set of clusters that are available to the tenant t is denoted as C_t , *i.e.*, $C_t = \{c | \hat{y}_t^c = 1, c \in C\}$, where $|C_t| = k$. When variables $\{\hat{y}_t^c\}$ have been determined, RGCM will assign a gateway cluster to each VPC $v \in V$, *i.e.*, determine variables $\{\hat{x}_v^c\}$. For each VPC $v \in V$, the algorithm selects a cluster $c \in C_t$ with the least burden and sets variable \hat{x}_v^c to 1.

While solving a linear programming might take a long time for a large network, we note that tenants/VPCs/instances are deployed incrementally, and the number of VPCs/tenants is usually much smaller than that of instances. For example, if hundreds of thousands of instances boot up at the same time, the corresponding VPCs are thousands and the corresponding tenants are hundreds. We utilize Gurobi solver [34] to run the RGCM algorithm on a server equipped with a 10-core Intel i9-10900 CPU. The solution time is 1.15s for the network with 10 gateway clusters, 568 tenants and 1885 VPCs in §8.4, which is acceptable compared to the VPC/instance deployment time.

B Additional Implementation Details

B.1 eBPF Map Size

In the current Linux kernel implementation, the memory usage of an eBPF hash map grows with its entry number. However, the maximum entry size is bounded by the `max_entries` defined by XDP/eBPF program during map initialization. The user space function `bpf_map__resize()` can resize an eBPF map only before it is initialized in the kernel. Unfortunately, we cannot resize an eBPF map after it is created. We have to deploy a new XDP/eBPF program to reinitialize the map size.

Thus, the number of instances that a gateway cluster can serve is limited by the `max_entries` of the eBPF maps. For example, the endpoint hash map in Zeta stores instance forwarding rules and its `max_entries` is set to $128*1024$ ($\sim 131k$). To avoid the above limitation, we can set a larger entry number for the endpoint hash map, such as $1024*1024$ ($\sim 1M$). In addition, the key size and value size of one

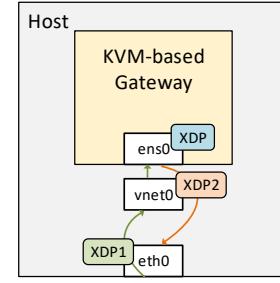


Figure 26: Early Version of KVM-based Gateway.

endpoint entry is 8 bytes and 16 bytes, respectively. The total memory size of 1M entries is only 24MB.

B.2 Best Practice for Physical Deployment

Zeta is usually deployed as two self-contained parts: (i) One Kubernetes micro-service hosting Cluster Controller services; (ii) One Gateway Cluster for Zeta data plane, which is based on physical machines in production environment.

Figure 25 illustrates the best practice of Zeta deployment, which includes a control node, several gateway nodes and compute nodes. The leftmost control node deploys the management service of the cloud platform and Kubernetes cluster hosting Zeta Controller. The middle ones are gateway nodes, each of which deploys DFT and FWD modules. The `eth1` interfaces of all nodes access the Device Management Network. In addition, we use separate interfaces for the Zeta API Network and Tenant Network, which prevents massive tenants' traffic from blocking the control messages. The Zeta API Network is responsible for sending the operation instructions and reporting status information, including OAM packets, IPs allocation/migration policies and gateways' load information. The Tenant Network transmits the east-west traffic through the VXLAN tunnel [48] for tenant instances.

B.3 Additional Details of Virtual Deployment

In the early development of Zeta, we use TUN/TAP device [74] as the NICs of KVM-based gateways. In addition to deploying XDP in the KVM-based gateways, we also deploy additional XDP programs on the physical machines to accelerate the host-VM datapath [5, 80]. As shown in Figure 26, we attach XDP1 to the NIC (*i.e.*, `eth0`) of the physical machine to accelerate the host-VM ingress traffic. We attach XDP2 to the TAP device (*i.e.*, `vnet0`) on the physical machine to accelerate the VM-host egress traffic.

However, Zeta suffers from the poor forwarding performance. For example, the packet forwarding rate of a KVM-based gateway equipped with 4 vCPUs is only 1.36Mpps. The reason is that attaching XDP program to VM's NIC will affect the function of TAP device in host and lead to a significant hit on VM RX performance [5].

Finally, Zeta adopts SR-IOV [23] for KVM-based gateways. Although the driver of Intel XL710 VF (i.e., iavf) does not support XDP Native mode, and Zeta adopts XDP Generic mode with reduced performance in KVM-based gateways [19, 36]. We obtain an acceptable forwarding performance. For example, the pure forwarding rate of one virtual core is 0.86Mpps under 2k entries, which drops 54% compared with one physical core with XDP Native mode.

C Additional Evaluation Details

C.1 Transformation of Google cluster-data

We query the `a.CollectionEvents` table of Google cluster trace and obtain the mapping of `<user, machine, job>` [30]. The machine number is 10001 and the user number is 1952. Considering that we only have 20 compute nodes, while there are 10001 machines in the table. Thus, we merge the jobs of every 500 machines to one compute nodes.

C.2 Linear Scaling Throughput of Gateways

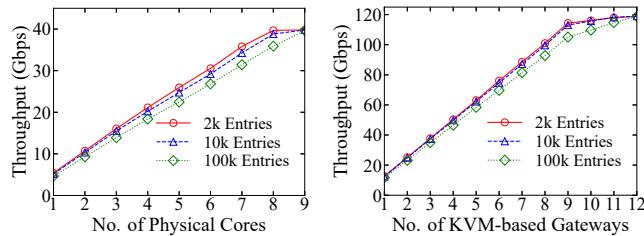


Figure 27: Throughput vs. No. of Physical Cores Figure 28: Throughput vs. No. of KVM-based Gateways

Linear Scaling Throughput. Figures 27-28 show that the total throughput will scale linearly with the increasing number of physical cores and KVM-based gateways. Specifically, when the inner packet size is 512 bytes and the number of entries in eBPF maps is 2k, the throughput of a physical core is 5.4Gbps, and 8 physical cores will hit the NIC’s bandwidth limit of the physical machine at 40Gbps. The throughput of a KVM-based gateway with 4 vCPU is 12.7Gbps, and 9 KVM-based gateways will nearly reach the NICs’ total bandwidth limit of the 3 physical gateway machines at 120Gbps. In addition, the timeout mechanism of maps for flow statistics leads to throughput degradation with the number of entries increases. We will try to optimize this issue in future work. The linear scaling throughput of Zeta gateways greatly enhances the scalability of Zeta gateway clusters.

Aquila: A unified, low-latency fabric for datacenter networks

Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, Amin Vahdat
Google Inc.
aquila-nsdi2022@google.com

Abstract

Datacenter workloads have evolved from the data intensive, loosely-coupled workloads of the past decade to more tightly coupled ones, wherein ultra-low latency communication is essential for resource disaggregation over the network and to enable emerging programming models.

We introduce *Aquila*, an experimental datacenter network fabric built with ultra-low latency support as a first-class design goal, while also supporting traditional datacenter traffic. Aquila uses a new Layer 2 cell-based protocol, GNet, an integrated switch, and a custom ASIC with low-latency Remote Memory Access (RMA) capabilities co-designed with GNet. We demonstrate that Aquila is able to achieve under 40 μ s tail fabric Round Trip Time (RTT) for IP traffic and sub-10 μ s RMA execution time across hundreds of host machines, even in the presence of background throughput-oriented IP traffic. This translates to more than 5x reduction in tail latency for a production quality key-value store running on a prototype Aquila network.

1 INTRODUCTION

There has been tremendous progress in datacenter networking over the past decade, with fundamental advances in the control plane [18, 27, 44, 49], the rise of commodity silicon arranged in non-blocking topologies [4, 23, 36, 49], network management and verification [7, 8, 29, 41], and highly available network design techniques [21]. Taken together, the community is now in a place where cost-effective, easy-to-manage, and scalable network designs and deployments are becoming common in industry. Plentiful network bandwidth at the scale of clusters of tens of thousands servers [49] can be leveraged for large-scale hyperscalers and the services they host.

However, all of these advances come while assuming TCP-based congestion control and Ethernet Layer 2 protocols. This Layer 2-4 stack has been incredibly robust and resilient through many decades of deployment and incremental evolution. However, we are seeing a new impasse in the datacenter [12] where advances in distributed computing are

increasingly limited by the lack of performance predictability and isolation in multi-tenant datacenter networks. Two to three orders of magnitude performance difference [15] in what network fabric designers aim for and what applications can expect and program to is not uncommon, severely limiting the pace of innovation in higher-level cluster-based distributed systems.

Such concerns are amplified when considering the state of supercomputing/HPC clusters [17] and emerging machine learning pods [22, 28] where individual applications benefit from low-latency RDMA [16, 51], collective operations [46], and tightly integrated compute and communication capabilities. The key differences in these more specialized settings relative to production datacenter environments include: i) the ability to assume single tenant deployments or at least space sharing rather than time sharing; ii) reduced concerns around failure handling; and iii) a willingness to take on backward incompatible network technologies including wire formats.

Recent research efforts into disaggregated rack-scale architectures [13, 34] further highlight some of these challenges: can the same NICs and switches supporting host-to-host communication across the wide area support, for example, SSD and GPU devices at a much smaller radius? Is the disaggregation network necessarily a separate dedicated fabric or can it be multiplexed with TCP/IP traffic destined to remote hosts potentially 100ms or more away? While there is some appeal to running a second (or third) network dedicated for an individual use case, the control and, as importantly, the management overhead of each network introduces a cyclic dependency where the second network is not worthwhile relative to the status quo until the underlying technology is proven/mature. However, there is no opportunity to iterate on the alternate technology because doing so is cost and complexity negative for a number of generations into the future because applications would have to evolve substantially before demonstrating end-to-end wins on the new hardware.

The need for backward compatibility combined with challenges in deploying niche "bag on the side" networks threatens a new ossification in datacenter networking and dis-

tributed systems where we are left with programming to the lowest common denominator of TCP transports and commodity Ethernet switches with associated latency, CPU efficiency, and isolation limitations.

In this paper, we present a first exploration of an alternative tightly-coupled (or *Clique-based*) datacenter architecture, *Aquila*, a hardware implementation supporting predictable, high-bandwidth, and ultra-low latency communication. In our approach, datacenter networks consist of dozens of Cliques, each hosting approximately 1-2k network ports. Cliques interoperate with one another at the datacenter interface (e.g., the spine layer of existing Clos-based datacenter networks) through standard Ethernet and IP. However, within a Clique, any transport and Layer 2 network protocol may be deployed. Applications that fit within the boundaries of an individual Clique can assume Clique-local capability, including robust RDMA, predictable low-latency communication, device disaggregation, support for ML aggregation primitives, etc. We assume IP-based transport for communication between Cliques, which means that any intra-Clique communication primitives and innovations must live alongside standard transports. Cliques then become the unit of deployment, innovation, and homogeneity, allowing for incremental, backward-compatible deployment into existing datacenters. A Clique is also sufficiently large to host all but the largest of individual distributed systems, especially as we move to hundreds of compute cores per server.

Aquila, our first Clique implementation based around a custom in-house ASIC and communication software, consists of a cell-switched non-Ethernet substrate, GNet.

- *Aquila* networks are built from individual silicon components that serve as both NIC and a portion of the traditional Top of Rack (ToR) switch; each *ToR-in-NIC* (TiN) chip attaches to hosts and directly to other TiN chips to realize a cost-effective network built from a single, replicated silicon component, rather than distinct NIC and switch silicon components from separate vendors.
- GNet provides the illusion of Ethernet to hosts within *Aquila*, as well as to non-*Aquila* networking components outside the scope of the *Aquila* Clique, by terminating Ethernet at the *Aquila* network boundary and tunneling traffic across a fully-custom, self-defending, near-lossless L2 substrate.
- *Aquila* further reduces cost by realizing a direct-network rather than an indirect (Clos) topology. To fully unlock the capabilities of its Dragonfly topology [30], and freed from the de facto constraints imposed by Ethernet, *Aquila* leverages adaptive routing to deliver full point-to-point bandwidth between host-pairs by leveraging multiple non-minimal paths.
- *Aquila* delivers data in small chunks called *cells*, rather than packets, thereby optimizing for latency of small exchanges like those used by distributed systems built on RDMA and similar technologies [51]. Its extremely tight

integration between NIC and network allows for ultra-low RMA-read capability (4us median) between the memory systems of up to 1152 hosts.

Aquila's design departs from traditional Ethernet fabrics in several ways: i) links use credit-based flow-control; ii) switch buffering is shallow; and iii) solicitation bounds end-to-end admission. Any one of these tenets in Ethernet would be problematic, but taken together, they form a cohesive design. For instance, flow-controlled near-lossless links can give rise to tree saturation, especially with shallow buffering, but end-to-end admission control bounds the size and spread of such trees, and ensures they are transient. Similarly, admission control breaks down when drops are likely, but link-level flow control makes drops very rare, and in turn enables the use of shallow buffering in switching elements, since overrun is not possible.

We present the detailed design, implementation, and evaluation of *Aquila*. *Aquila* is not the final word in Clique design; in fact, our first experience with the *Aquila* system suggests a number of areas for improvement in future generations. We hope, however, that the approach of bringing vertical integration including the host software stack, the NIC, and the switch along with a Clique-based datacenter architecture will enable new models of datacenter innovation along with new capabilities to distributed systems that can assume cutting edge rather than lowest common denominator communication and disaggregation capability within the boundary of thousands of servers and hundreds of thousands of cores.

2 OBJECTIVES AND OUR APPROACH

Aquila's design departures from Ethernet are grounded in a set of common objectives, described below. Taken individually, these design choices—e.g., flow control, custom Layer 2—would be hard to apply to an existing network incrementally. But in concert, *Aquila*'s features realize a complete, performant design point.

Sustainable hardware development. To sustain the hardware development effort with a modest sized team, we chose to build a single chip with both NIC and switch functionality in the same silicon. Our fundamental insight and starting point was that a *medium-radix* switch could be incorporated into existing NIC silicon at modest additional cost and that a number of these resulting NIC/switch combinations called *ToR-in-NIC* (TiN) chips could be wired together via a copper backplane in a *pod*, an enclosure the size of a traditional Top of Rack (ToR) switch. Servers could then connect to the pod via PCIe for their NIC functionality. The TiN switch would provide connectivity to other servers in the same Clique via an optimized Layer 2 protocol, GNet, and to other servers in other Cliques via standard Ethernet. The inset in Figure 1 summarizes the major components of TiN.

Cost effective, non-blocking topology. For efficiency and low latency, we selected a direct connect topology, Dragonfly, a well-studied topology that minimizes the number of

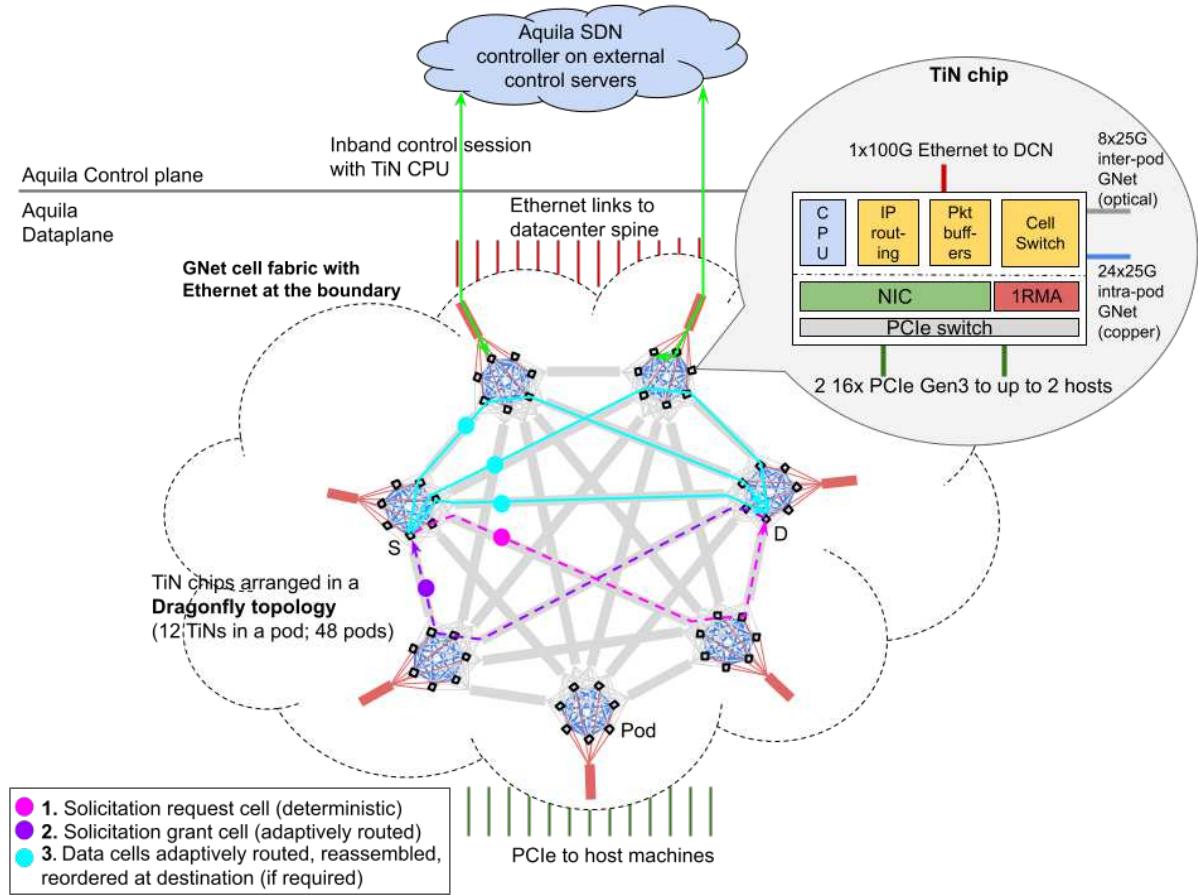


Figure 1: Aquila Clique dataplane and control plane overview. The ToR-in-NIC (TiN) chip is expanded in the top right inset. TiN chips are arranged in a cell-switched Dragonfly topology (12 TiNs in a pod; 48 pods). Conventional Ethernet/IP packets are split into cells at ingress after a round of solicitation per packet and reassembled and re-ordered at the fabric’s egress. The co-designed 1RMA protocol injects cells directly into the cell network, extending memory accesses across the Clique. The Aquila SDN controller configures and manages TiN switches inband, via the DCN.

long optical links in the network while still providing non-blocking bandwidth for uniform random traffic patterns, with 2:1 over-subscription for worst-case adversarial traffic. Figure 1 illustrates a simplified Dragonfly topology where TiNs within a pod are fully connected in a mesh, and multiple pods are likewise connected all-to-all to form a tightly-coupled Clique network. The largest Aquila network supports 12 TiNs in a pod with 48 pods, serving up to 1152 host machines.

Combining NIC and ToR into the single TiN chip was a less costly path to innovation than separate NIC and switch ASIC programs, and a design realized from a common single component was intended to streamline inventory management for Aquila. Further, we implemented an optional capability to allow pairs of host machines to share a single TiN, halving the normalized cost of ownership for networking per host, trading off reduced sustained bandwidth provisioning per machine.

Ultra low-latency network. To optimize for ultra-low latency, under load and in the tail, Aquila implements cell-based communication with shallow buffering for cells within the network, flow controlled links for near lossless cell delivery, and hardware adaptive routing to react in nanoseconds to link

failures and to keep the network load balanced even at high loads. To ensure recipients are not overwhelmed, Aquila implements end-to-end solicitation for each packet at ingress, which guarantees that resources are available at the destination TiN before the packet can be split into cells and transmitted from the source TiN. We built these latency-guarding features into Aquila’s Layer 2 protocol, GNet. As depicted in Figure 1, while the Aquila network fabric presents an Ethernet packet interface at its boundary, Aquila tunnels conventional Ethernet/IP packets over GNet, disassembled at ingress and reassembled and re-ordered at the egress of the Clique.

Unified fabric for legacy traffic and RMA/memory disaggregation. Aquila unifies low-latency communication primitives (RMA) alongside commodity primitives (IP) in a common fabric, to address the growing diversity of datacenter workloads [5, 42, 45]. A fabric delivering both high-performance and legacy connectivity avoids the pitfalls of a *bag-on-the-side* network and secondary NICs, reducing the cost of ownership and the toil related to the life cycle management of two separate networks. Managing a single network for availability, security, monitoring and upgrades

is challenging enough—managing separate networks for individual use cases introduces an extraordinarily high bar in any cost/benefit analysis. For efficient remote memory access and memory disaggregation alongside traditional protocols, we co-designed a Remote Memory Access protocol, 1RMA [51], to extend memory access across the Aquila Clique directly on GNet, instead of layering on top of IP.

Co-existing within the larger Clos-based software-defined datacenter network ecosystem. Typical datacenter networks [49] are based on a scalable Clos topology where aggregation blocks are connected via a spine switching layer; Aquila is designed to integrate into such a network via its Ethernet ports. A hierarchical Software Defined Network (SDN) control plane with a modular, micro-service architecture [18] manages and controls the various networking blocks within the datacenter.

Figure 2 describes the integration of Aquila in the broader datacenter network’s dataplane and control plane ecosystem. The Aquila network block connects to the datacenter’s spine switching layer via Ethernet links, akin to other aggregation blocks. The modular architecture of the datacenter network realizes a hybrid topology, i.e., a Dragonfly network integrated as a block within a larger Clos topology, a first of its kind to the best of our knowledge.

For the control plane, we adapted an SDN controller to configure, manage and program TiNs inband via a thin on-box firmware running on the TiN CPU (Figure 1). The Aquila SDN controller, similar to the SDN controllers of other aggregation blocks, interacts with each of four central Inter-Block Routing Controllers (IBR-C) (Figure 2) to enable communication with other aggregation blocks as well as with networks external to the datacenter.

Cliques as the basis for hosting tightly-coupled applications. To exploit the tightly coupled, low latency communication enabled by the Aquila Clique, we adapted the job scheduler [53] to be aware of Clique locality. High bandwidth or latency sensitive jobs could optionally be scheduled on host machines within a Clique, while other jobs could still be bin-packed across blocks, regardless of locality.

3 HARDWARE DESIGN

In this section we relate how the key design goals drove the hardware design. In summary:

- **Low latency** objectives drove the selection of a shallow-buffered cell-switched GNet fabric. §3.1 details the design of the GNet switch and link-level protocol.
- **Cost-effectiveness** goals led to the choice of an integrated switch and NIC chip, TiN, as well as a direct topology such as the Dragonfly. §3.2 outlines the rationale for selecting the Dragonfly and the impact of this choice on the design.
- **Shared fabric for both IP traffic and low-latency RMA.** §3.3 describes how IP packets traverse the GNet fabric, and §3.4 details the co-design aspects of the 1RMA protocol with the GNet fabric.

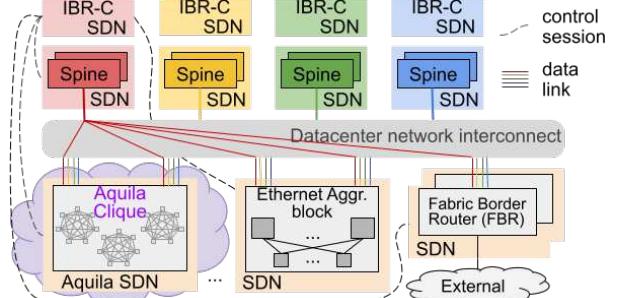


Figure 2: Aquila Clique integrated into the broader datacenter network and SDN ecosystem co-existing with other Ethernet aggregation blocks. Topologically, the Aquila block connects to the Clos-based datacenter spines akin to the Ethernet based blocks. In the control plane, the Aquila SDN controller, similar to controllers of other blocks, interacts with each of the four *sharded* Inter Block Routing controllers (IBR-C) to enable cross-block routing.

3.1 GNet Switch and Links

The cell switch. The switching capability of the TiN chip is provided by a 50-port cell switch optimized for low latency. The maximum cell size of 160 bytes was chosen to keep the serialization latency on 25G links small (~50ns). 32 ports are external-facing GNet ports (of which 24 are pod-local and 8 are inter-pod ports). The remaining 18 ports are intra-chip, for cells transmitted and received by the various traffic endpoints (e.g., IP and 1RMA). The fall through latency of the core cell switch is 20ns and the total per hop latency without Forward Error Correction (FEC) is 40ns. GNet links support 32 Virtual Channels (VCs [14]) - FIFO queues used for deadlock-avoidance and QoS. VCs are used for deadlock-free routing, for differentiation between classes of service, and to separate solicited and unsolicited traffic. A centralized arbiter implements a variant of the iSLIP arbitration protocol [38], supporting one arbitration request per VC per port, ensuring that no VC or port is starved of throughput. To support variable cell sizes, we modified iSLIP such that the ingress and egress ports communicate a "busy" signal to the crossbar arbiter. A "busy" indicates that the ports are transferring a cell across the crossbar. The arbiter takes this into account when it evaluates pending requests for the next request-grant-accept cycle. Quality of Service (QoS) between VCs is implemented in the output buffer and supports both weighted round robin and strict priority. Each VC has its own input FIFO space protected by a reliable credit mechanism, similar to that used in PCI Express. A shared buffer, shared credit scheme was considered to save memory, but for the relatively short links required for Aquila the simplicity and complete QoS isolation of independent FIFOs was preferred.

GNet link level protocol. GNet links support cells between 16 bytes and 160 bytes in size, with frequent reverse flow control traffic. The use of variable length cells gives very high protocol efficiency (e.g., 1RMA requests are small) on the wire even after the additional control traffic for admission control. Every GNet cell has a common routing header of 8 bytes that contains the 16 bit source and destination GNet ad-

addresses, the cell length and type, the VC, a decrementing hop count, an 8 bit header CRC and a Trace Enable bit. To enable efficient transmission of GNet cells, a custom 66/64 bit Physical Coding Sublayer (PCS) was developed that minimizes the cell delineation overhead and allows the reverse flow control traffic to be sent as very compact ordered sets. Control ordered sets are used for: (1) Start of cell delineation, (2) Flow control, (3) TimeSync (§B), (4) Management ordered sets (MOS), and (5) Phy up/down control and fault detection.

3.2 The Dragonfly cell fabric

We selected the Dragonfly topology to manage the cost of optical links, shown in Figure 1. The Dragonfly cell fabric is implemented using two types of GNet links: 24 *local* links per TiN that fully connect TiNs within the pod, and 8 *global* links per TiN that connect between pods, up to 100m apart on the datacenter floor. The local links are implemented as single-lane, 28 Gbps copper backplane connections. The global links are optical and use specially developed low cost GNet optical modules. Noise on global links is mitigated with FEC, incurring a 30ns per-hop latency penalty, and a 6% bandwidth overhead. Local links operate without FEC for the lowest latency at acceptable margins. Both local and global links ultimately implement the same link level protocol. Due to the hierarchical nature of the Dragonfly topology, GNet addresses have three components: *pod id*, *TiN id* and *endpoint id*. Endpoints represent protocol engines (IP, 1RMA, and CPU) detailed later.

Deadlock avoidance. We implement deadlock avoidance in our Dragonfly topology using a combination of turn rules and VCs. With our budget of 32 VCs, it is desirable to minimize the number of VCs used for deadlock avoidance. In the implemented routing scheme, we employ turn rules similar to the *parity-sign* approach in [20] within a pod for deadlock-free intra-pod routing. The VC is incremented when moving from a global link to a local link [30], requiring a total of 3 VCs used for deadlock avoidance in the worst fault-free route, that of a non-minimal route via an intermediate pod. Accounting for 10 traffic classes, each with 3 routing VCs, a further two VCs are available as *escape VCs* in certain dynamic failure avoidance scenarios.

Adaptive routing. The majority of traffic routes adaptively to achieve both high throughput and the lowest latency on Aquila’s Dragonfly network. TiN implements locally adaptive routing [30, 48], a scheme that makes adaptive routing decisions based on available information at a GNet switch, in particular, the per-VC output queue lengths at each port. These queue depths reflect nearby congestion because of GNet’s link-level flow control and shallow buffering. Link failures manifest similarly, which also allows the adaptive routing algorithm to route around failed links until the SDN routing engine removes the entries for links which have lost connectivity.

The adaptive routing implementation selects two minimal routes at random from eight supplied by the routing tables, and

also considers three non-minimal routes from 24 non-minimal route candidates. The five candidate routes are evaluated using a weighted comparison that favors the minimal routes. Random choices (rather than 24-way comparison) allow us to avoid flocking, having coordinated adaptive routing decisions, and moving congestion from one place to another [40]. Other routing modes are enabled by constraining the routing to minimal routes only, or by forcing deterministic choice of route using a hash of the source and destination addresses. These constraints yield Aquila’s four principal routing modes: Fully Adaptive, Minimal Adaptive, Deterministic and Minimal Deterministic. The deterministic routing modes are used for cell types requiring ordering. The cell switch uses table-based routing because of the need to handle failures and upgrades using SDN routing described in §4.2, as well as flexibility for other topologies.

3.3 IP Traffic

Host IP traffic is sent and received by a conventional 100 Gbps NIC capable of supporting multi-host operation for up to two independent hosts. The option of multi-host capability was considered important in order to give a degree of flexibility in bandwidth per machine allocation. There are two other sources of IP traffic on the TiN chip: the 100 Gbps external Ethernet port for connectivity outside the Aquila fabric, and a low bandwidth port to the embedded management processor. Traffic from all IP sources is handled in the same way: the packet processing pipeline performs IP routing and cellification, i.e., splitting the IP packet into GNet cells and traversing them to the final destination, using Aquila’s IP over GNet protocol.

Packet processing logic. Each IP packet passing over the GNet fabric goes through the input packet processing and output packet processing blocks once only. Effectively, the entire GNet Clique acts as a single stage IP packet switch. The input packet processing pipeline handles:

- L3 to GNet L2 address translation (either one-to-one or WCMP [55]);
- Selectively punting some packets to the embedded control processor;
- Input buffer QoS.

L2 Ethernet MAC addresses are stripped from inbound packets after processing; packet transfer over the GNet cell network is for IPv4/IPv6 only. Non-IP packets such as ARP may be either encapsulated or punted to the embedded control processor, consistent with the requirements of our SDN control plane (§4).

IP over GNet protocol. IP traffic traverses Aquila by means of the GNet upper layer protocol, shown in Figure 3. Each IP packet sent over the cell fabric issues a *Request To Send* (RTS), and awaits a *Clear to Send* (CTS) handshake before any data is transmitted. These are sent as 16 byte GNet cells to minimize the bandwidth overhead. The handshake protocol performs three functions:

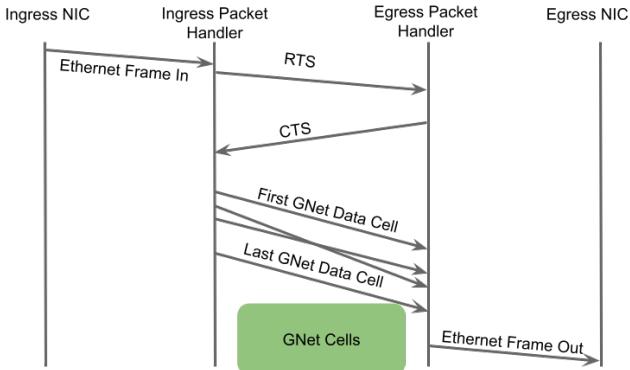


Figure 3: Classification: IP Packets are split into multiple GNet data cells that are only admitted into the GNet fabric when the ingress TiN receives a CTS. In the egress packet handler, cells are reassembled into packets, respecting their original transmission order, and sent to the NIC (all within the TiN chip).

- It implements solicitation for IP packets by only allowing data cells onto the GNet network when the destination end point has signalled it has sufficient input bandwidth and buffer space to receive them.
- It allocates hardware resources at the destination, e.g., cell to packet reassembly buffers, before any data cells are transmitted so that there is always a guaranteed reassembly space.
- RTS arrival defines inter-packet order. While data cells route adaptively and potentially arrive out of order at their destination, RTS ordering ensures that original transmission order can be reconstructed at the receiving side.

Packets which have passed through the packet processing pipeline and have a valid GNet L2 address are stored in the packet ingress buffers. An RTS is generated immediately; in fact, for long packets the RTS can be issued before the whole packet is received. The RTS itself consists of 8 bytes of routing header and a further 8 bytes of payload that includes the IP packet length, Class of Service (CoS), the packet's location in the ingress buffer, and an indicator of ingress buffer usage. Compactness is important because RTS cells are unsolicited and can still lead to incast. However, considering that an average packet is >1Kbytes, an incast of RTS cells represents a reduction of incast volume in the network by a factor of 64.

RTS cells are carried on their own VCs, allowing them to be sent at high priority and also maintain isolation between solicited and unsolicited cells. RTS VCs are routed deterministically over the GNet fabric, using a path selected by a hash of the flow-invariant fields of the cell, ensuring that the RTS cells for a given IP flow are received in the order they were sent. The RTS cells are received into FIFO queues at the packet egress. Packet data transfer is initiated by the egress-side by sending a CTS back to the appropriate ingress port. Along with the 8-byte routing header, the CTS carries a pointer to the packet in the ingress buffer (copied from the RTS) and a pointer to the allocated location in the egress cell-to-packet

reassembly buffer. CTS cells are issued by the CTS scheduler, which tracks the availability of egress reassembly buffer capacity, only issuing a CTS when there is space available to reassemble cells into packets.

When a CTS is received back at the packet source, the packet in question is pulled from the ingress buffer as a series of data-only cells, which are then transmitted across the fabric. Data cells can take many different routes (adaptively) through the fabric, and data cells may arrive in any order at the final destination. Cells are reassembled into packets in the egress buffer at the destination. The sizing of the egress buffer is determined by the bandwidth delay product of the output port bandwidth and the cell fabric round trip delay, plus an allowance for packet reordering delays. Packets do not experience significant queuing in the egress buffers, which are primarily for reassembly, so the egress buffers are significantly smaller than the ingress buffers.

In order to maintain packet order within flows, when a CTS is issued by the scheduler, the packet descriptor is registered with packet reordering logic respecting RTS arrival order. A packet is transmittable at egress after receipt of all its data cells, but transmittable packets are held until all packets in the same flow that were ahead of it in CTS issue order have been successfully forwarded to the NIC.

A significant benefit of the RTS/CTS scheme is that the RTS queues have a local view of all the requested packet demand for that destination port from the entire GNet fabric, while the packet *data* remains queued in the ingress buffers. In the presence of severe incast, packets can be discarded while conserving fabric bandwidth, i.e., without packet data traversing the cell fabric. The egress side can choose to drop a packet by issuing a variant of CTS (a *Clear To Drop*, CTD), which pulls the packet from the ingress buffer and discards it. A CTD is sent when an RTS is received at an RTS queue whose depth exceeds a given threshold. The RTS queue's depth also provides the signal for Explicit Congestion Notification (ECN) marking; if the RTS queue exceeds the marking threshold when the packet has been reassembled and is ready to send, ECN is applied.

QoS Support for IP. There are separate RTS queues for each independent port and class of service, with a total of 32 RTS queues supporting eight CoS on four independent ports - one port for the external Ethernet MAC, two for the dual-host NIC, and one for the control processor. CTS cells are issued by the CTS scheduler at the packet's destination which allocates bandwidth between the 32 RTS queues, implementing per-IP-packet QoS between the respective queues. The CTS scheduler may throttle traffic into the egress buffers by limiting CTS issues according to a window of outstanding packet fetches, which can be adjusted to minimize the queuing of data cells within the cell fabric. The scheduler does not attempt to implement bandwidth fairness between sources since all the sources to a given destination port share the same RTS queue.

3.4 1RMA

To deliver the low-latency capabilities of the Aquila Clique directly to distributed systems programmers, we built an implementation of 1RMA [51] into the TiN chip. 1RMA is an RMA protocol that offers unordered, segmented, solicited remote memory access primitives (read, write, and atomics) to on-host software—tenets that match precisely those of GNet packet transfer governed by RTS/CTS.

Such alignment is not merely coincidental; we co-designed Aquila and 1RMA’s GNet-based protocol. Rather than simply layering the 1RMA protocol messages above the packet layer, we instead express 1RMA protocol exchanges as first class cell types in GNet—alongside RTS and CTS, rather than atop—and ensure that they obey similar end-to-end solicitation rules as they share the Aquila fabric. The advantage of co-design is significant latency savings: while a UDP/IP or TCP/IP round-trip on Aquila incurs *six* GNet half-round-trips on its critical path (RTS, CTS, data, in each direction), a 1RMA read operation incurs only *two*, shaving precious microseconds from user-facing latency.

We realized protocol co-design by encoding 1RMA *read requests* entirely within GNet framing. Fundamentally, read requests initiate data transfer from receivers to senders, i.e., such requests intrinsically already are solicitations, expressed at the transport layer. GNet also builds on solicitation, but at the L2 layer. The key insight is to express both the GNet (L2) and 1RMA (L4) solicitation behaviors in a single cell type, *Req*. Since Req cells solicit data movement in the reverse direction, GNet handles Req similarly to CTS; the main differences arise from cell size, as Req fully encodes a read request (host address, memory identifiers, HMAC, etc.), yielding a cell 3x larger than CTS at 48B. Req is otherwise behaviorally similar to CTS, in that it can be freely reordered without violating assumptions of the protocol layer above. Because 1RMA is highly tolerant of out-of-order delivery, Req is intrinsically compatible with Aquila’s adaptive routing.

We also leverage 1RMA’s close coupling to host-facing PCIe to encode response cells, *Resp*. 1RMA NICs send each individual PCIe read completion payload as a distinct protocol response, a hardware simplification that avoids response coalescing logic, buffering, and overheads in the NIC. To facilitate this behavior in GNet, Resp cells are sized to handle the most common PCIe completion sizes we observe from the host root complexes. Like Req, Resp can be freely reordered and routed adaptively, and the initiating 1RMA NIC lands the individual response segments in arrival order in destination host memory, since there is no need to restore overall inter- or intra-request response ordering.

Lastly, to isolate latency-critical 1RMA traffic from less sensitive IP flows, we map roughly half of GNet’s virtual channels to carry low-latency protocol messages, which 1RMA shares with low-latency IP traffic flows. Because IP traffic is cellified, 1RMA responses do not queue behind bulk transfers from competing flows. In all, 1RMA on Aquila de-

livers near-flat lookup latency—even under load from conventional traffic—to approximately 864TB of DRAM inside of 4us end-to-end. Aquila traversal accounts for a mere 2.5-3us; the remaining time is attributable to PCIe latency contributions.

3.5 Embedded control processor

The TiN chip has an embedded control processor (ECP) to handle all switch side control and monitoring actions. Cost of silicon exerts pressure to make the ECP as simple as possible, as it is replicated in each TiN chip. Where a typical control processor for a ToR might be a multicore, 64-bit processor with 8-16 GB of memory, TiN’s ECP is a 32-bit ARM Cortex M7 processor with a mere 2 MB of SRAM.

In order to bootstrap the embedded control processor before the GNet logic has been fully initialized (§4.4), a low bandwidth but reliable in-band control path is implemented over the GNet fabric using the management ordered set (MOS). Each MOS 64 bit word allows 6 bytes of data to be transferred between directly connected TiN chips, irrespective of whether the GNet link layer is up. We layer a robust packet implementation, PMOS, above the MOS primitive to carry debug and bootstrap traffic.

3.6 Putting it all together

Figure 4 plots the overall structure of the TiN chip:

- The cell switch (the building block for the cell fabric);
- A conventional IP host interface (NIC);
- An external-facing Ethernet MAC for connectivity to outside networks;
- A 1RMA host interface that supports direct protocols across the cell fabric;
- IP packet-to-cell (ingress) and cell-to-IP packet (egress) logic;
- The embedded control processor, acting as the local agent for the SDN control software.

The device has the following interfaces:

- Two x16 PCIe gen 3 interfaces giving 256 Gbps connectivity to one host or 128 Gbps to each of two hosts;
- Thirty-two 28 Gbps, single-lane GNet links used to construct the low latency cell fabric;
- A single 100 Gbps Ethernet interface which connects to the wider datacenter network (DCN).

Approximately 50% of the TiN silicon area is used to implement host interface or NIC functions, and 50% for switching functions.

4 SOFTWARE-DEFINED NETWORK

As alluded to in §3, the integration of switch and NIC in a single chip leads to substantial replication of the management subsystem across all TiNs in an Aquila Clique. To keep the Aquila Clique cost-effective, the management subsystem for a TiN ASIC was kept simple – a 32-bit ARM Cortex M7 processor with a modest 2MB of SRAM and no dedicated management Ethernet port. Consequently, much of the routing

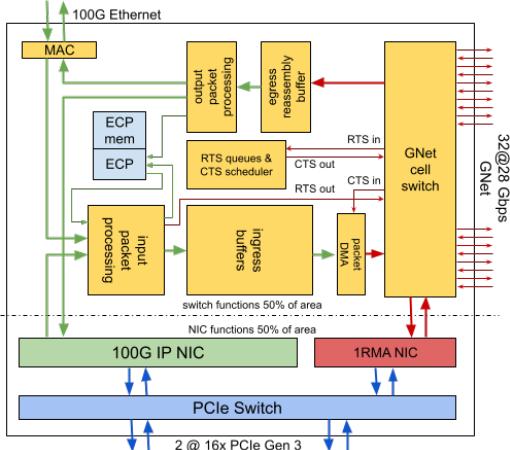


Figure 4: Aquila Chip Architecture showing GNet, IP, Embedded Control Processor (ECP), 1RMA and PCIe components.

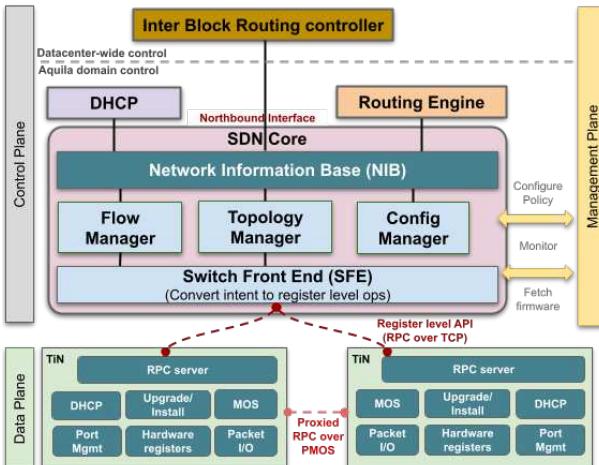


Figure 5: Overview of Aquila’s SDN and firmware architecture.

computation and state needed to be offloaded from the switch to a logically centralized distributed controller which had to orchestrate the bootstrap, management and control of the fabric in-band. In this section, we describe how Aquila’s software-defined network (SDN) control plane, along with its simplified firmware, was able to address the challenges of controlling and managing an Aquila Clique, specifically:

- Explosion of flow state in the SDN (§4.2).
- Switch firmware with constrained CPU/memory (§4.3).
- In-band bootstrap of the whole network (§4.4).

4.1 Control Architecture Overview

The Aquila controller is built on top of an SDN controller platform [18], a modular SDN control plane comprised of micro-services, and a central publisher/subscriber database called the Network Information Base (NIB). Multiple applications form an Aquila SDN constellation with redundant instances of each application deployed on separate control servers. The top half of Figure 5 details the Aquila SDN controller applications.

The routing application for the controller, Routing Engine (RE), computes the routing solution for the Aquila network

block in reaction to changes of topology states and external reachability. RE writes the solution in the form of flows and groups similar to OpenFlow [39] to the NIB in sequenced batches for hit-less routing state transition. Separately, the Inter-Block Routing controller (IBR), an application in a data center-wide SDN control domain, computes the routing solution for traffic between various network blocks and provides Aquila’s RE with the egress paths to reach destinations external to the Aquila Clique.

On receiving routing updates from the NIB, Flow Manager (FM) sorts the flow and group programming operations. For instance, a flow is installed only after its referenced group is installed for hit-less transition, before sending them to the Switch Front-End application (SFE) via RPC. SFE programs the flows and groups to TiN switches converting between flows/groups and hardware register values and completes the RPC with the programming results. Then FM writes the results back to the NIB for RE to consume.

4.2 Handling routing state

The large number of GNet endpoints in the fabric and the per-port GNet routing table in the TiN switch result in much larger routing state than non-Aquila blocks, which increases both CPU and memory demand in the SDN system. Aquila routing introduced scaling challenges for both IP and GNet flows.

IP flows. A network comprised of 1152 hosts and 576 management CPUs, addressable via both IPv4 and IPv6, calls for approximately 1.9 million flows, each with a single output port. Leveraging the observation that all of these flows are from a small number of subnets, we introduced a new *indexed* group representation, where the index of a port in the group corresponds to the same index in the subnet, which in turn reduces the number of flows by a factor of 576 (the number of TiNs in a fabric).

GNet flows. As seen in §3, flow controlled GNet requires per-input port, per-virtual channel flows which leads to an explosion in state for a switch with 50 ports. A naive implementation leads to almost 5 million flows. To accommodate such a large scale, we exploited the significant similarity in the routes. For example, all terminal ports described in §3.1 use the same route, and are represented only once in the NIB. Similarly, the deadlock avoidance turn rules define a similar role for each intra-pod port in the TiN chip. Further, all inter-pod ports behave the same. We introduced six *port classes* – denoting equivalence classes of ports with respect to routing rules – reducing the number of flows to approximately 700k. GNet flows use port-classes as both matching fields and output actions. On receiving a GNet flow using a port-class, SFE expands it to flows targeting each member port’s GNet flow table, and then prunes improper member ports from the output, e.g., to avoid sending traffic back to the source. The resulting flows are then programmed in the switch.

Even with these optimizations in place, the rest of the SDN system needed more modifications:

- Despite the port-class optimization, the number of flows in the NIB was still about 10x more than non-Aquila network blocks. To compensate for the memory increase, the NIB’s pub-sub interface was changed to keep state in compressed format and decompressed only when necessary.
- The SRAM available in the TiN switch is not large enough to hold a snapshot of all routing state. SFE has the capability to rate limit the hardware programming operations to avoid the memory on the switch from overflowing. The RPC interface between SFE and switches is designed in such a way that the largest RPC can fit in memory and only one outstanding RPC is allowed at a time.

4.3 Switch Firmware with limited state

The switch firmware (see lower half of Figure 5) runs on an ARM Cortex M7 CPU integrated into the TiN switch chip. Due to physical size and cost limitations, the firmware has only 2MB of on chip SRAM available. Therefore, it is built on the FreeRTOS [1] and lwIP [2] open source libraries to fit within the space constraints. The firmware is implemented in approximately 100k lines of C and C++.

We explicitly decided that the firmware is not responsible for fully configuring the TiN chip. At power on, the firmware brings up the GNet and Ethernet links and attempts DHCP over Ethernet. This enables the controller to connect early during initialization and finish the necessary configuration to allow the TiN chip to start passing traffic (for details see §4.4).

The programming API exposed by the firmware is low-level and allows the SDN controller to directly access hardware registers. The API is generated from the hardware register description and permits the SDN controller code to use symbolic names of the chip registers for convenience. Statistics and counters from the TiN chip can also be reported using the low level API. The SDN controller is able to configure a set of registers that should be periodically reported by the firmware. One of the programming API sets up ARP/NDv6 responses in reaction to requests from the attached machines so that the IP-to-MAC resolution could function properly even if the firmware loses connection to the SDN controller.

The firmware supports *Non-Stop Forwarding* (NSF) reboots to minimize disruption caused by upgrades and unexpected software errors. During reboot the firmware avoids changing any configuration that might impact traffic. Since the inband connectivity is not disrupted, the controller is able to quickly reconnect after a reboot without going through the bootstrap process. The implementation of NSF reboot was simplified due to the register level API since there is no need to save and restore state information, because the TiN chip maintains all the controller visible state during reboot.

While the firmware itself is stateless, the TiN chip and SDN controller are not. After any loss of connection between the firmware and SDN controller a process of reconciliation has to be initiated to resolve any differences between the hardware

registers and the SDN controller intent. These differences can occur if any commands were lost when the connection failed.

4.4 In-band Control and Bootstrap

A key challenge in Aquila’s SDN control was that the control channel from the SDN controller to the Aquila switches is in-band. This means that the controller needs to communicate with the management CPU of a TiN before it can program the routing tables of the TiN. During bootstrap, the controller sets up TCP connections in-band over the datacenter network to all TiNs in the Clique in iterative “waves”, configuring and programming routing tables as it gains control of TiNs in each subsequent wave.

Figure 6 shows k Aquila pods connected via intra-pod copper GNet links as well as inter-pod optical GNet links. Some TiNs (e.g., TiN 1, TiN 3 in Pod 1 and Pod k) are also connected to the spine layer of the datacenter via Ethernet datacenter network (DCN) links. We refer to these TiNs as *DCN-connected*. The Aquila SDN controller—running on external control servers—is initially reachable only over the DCN links. The TiN firmware sends DHCP discover messages over the DCN links if available. These DHCP messages are relayed by the spine switches to the DHCP server, which then assigns an IP address to the TiN management CPU based on the TiN MAC address.

The Aquila controller has records of the IP addresses intended for each TiN’s CPU from its own configuration. The controller continually attempts to connect to each TiN CPU via TCP session using its assigned IP address and a well known L4 port number. Once the IP address is known to a TiN’s firmware, a controller message destined to that IP is trapped by an ACL rule installed by the firmware and reaches the firmware. The response Ack is sent out the same interface the packet came in from thus enabling a TCP connection between the controller and the switch CPU of the DCN-connected TiNs. The controller can then configure the DCN-connected TiN and program its routing tables.

Once the controller establishes a TCP session with a DCN-connected TiN, it uses that TiN as a *proxy TiN* (e.g., Pod 1, TiN 3) to bootstrap a directly connected *target TiN* (e.g., Pod 1, TiN 2) using the point-to-point low bandwidth Packet Management Ordered Sets (PMoS) protocol (§3.5) between TiN CPUs. A target TiN—not yet configured with its IP address—also sends DHCP discovery messages over MOS over all GNet links, which are trapped by the proxy TiN and sent over its own session to the controller. The controller in turn relays the discover message to the DHCP server, and likewise relays the DHCP response, so that a target TiN learns of its assigned IP address indirectly. The controller proceeds to configure and program the routing tables in the target TiN via the proxy TiN over PMoS. After enough routing state is programmed, the controller can establish a TCP connection to the target TiN via the GNet routing pipeline and the proxy TiN (Pod 1, TiN 3) can then be used in turn to bootstrap yet another target TiN (e.g., Pod 1, TiN 4). Once a TCP session is established

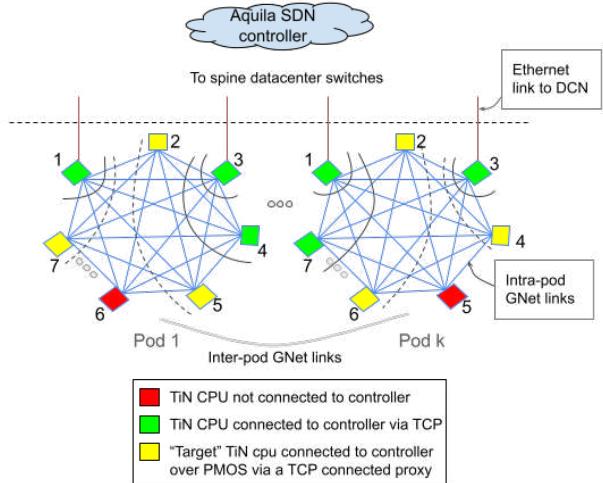


Figure 6: Inband bootstrap. The Aquila SDN controller bootstraps the TiNs inband in “waves” originating from the TiNs that are directly connected to the DCN.

with this new target TiN, it too can be used as a proxy to bootstrap a directly connected TiN (e.g., Pod 1, TiN 5), and so on.

DCN-connected TiNs typically bootstrap faster than the target TiNs, which are configured over the slower PMOS protocol leading to a distribution of bootstrap times ranging from 3 minutes to 48 minutes. Several of the waves of bootstrap occur simultaneously, resulting in a bring-up time of approximately 2.5 hours for a full-sized Clique.

5 EXPERIMENTAL RESULTS

We present a set of results examining key aspects of the Aquila network, including its data plane performance as well as its impact on application metrics.

5.1 Data Plane Performance

Aquila’s data plane performance was evaluated in a prototype Aquila testbed comprised of 576 TiNs. We used 500 host machines. Two hosts share a NIC unless otherwise specified. We used two workloads, both of which run with delay-based congestion control [33]:

1. *UR*: An IP traffic generator based on a user space micro-kernel, *Pony Express* [37], that generates a Uniform Random traffic pattern with Poisson arrival.
2. *CliqueMap*: A key-value store [50] that uses Remote Memory Accesses (RMA) via either Pony Express or 1RMA.

For our evaluation, we used three metrics:

1. *IP Fabric RTT (μs)*: We used NIC hardware timestamps to measure Aquila fabric RTT, excluding processing and ack-coalescing delays on the remote host. This is a true measure of the transmission and queuing delay inside the Aquila fabric, both for GNet and IP components.
2. *1RMA Total Execution Latency (μs)*: the time from when the RMA command is submitted to the hardware until the hardware issues the completion for that command.

This metric measures more than queuing and transmission delay in the fabric, as it includes the PCIe transaction delay on the remote side.

3. *Achieved throughput* of the network in Gbps (averaged over 30 seconds).

Latency Under Load. We examine the latency of both IP and 1RMA traffic under load. We used a CliqueMap client benchmark that issues lookups of 4 KB-sized values using RMA. By varying the QPS of the CliqueMap client on the 500 hosts, we changed the offered load per machine in a traffic pattern akin to Uniform Random. Figure 7 plots fabric RTT against offered load. It shows that the fabric latency remains under 40 μs, even when the network is close to the per machine NIC line rate of 50Gbps and it is sub-20 μs at 70% load.

1RMA is co-designed with GNet (§3.4) and Figure 8 shows that this co-design paid off with total execution time below 10 μs even under high load for 4 KB RMA reads that are generated using 500 CliqueMap clients to read from 500 CliqueMap backends.

1RMA Latency Isolation. Aquila is a unified network shared by low latency 1RMA traffic and regular IP traffic which may be latency insensitive. In our next evaluation, we show that Aquila delivers latency sensitive traffic with low tail latency despite sharing the network with IP traffic. To this end, we compare the latency of latency-sensitive traffic with and without background IP traffic in both Aquila and a conventional Ethernet network.

For the Ethernet network, we employ standard QoS techniques to isolate low-latency (or important) traffic from bulk throughput oriented traffic. We run 200 instances of CliqueMap lookups of 4 KB values at 10,000 QPS on a higher priority QoS class (H) and a UR traffic pattern with 64 KB messages with average load of 10 Gbps on a lower priority class (L). The relative egress scheduling priority between H and L classes is 8:1. The orange and cyan bars in Figure 9 show that such QoS-based schemes provide reasonable isolation for the CliqueMap traffic from the bulk IP traffic, leading to a modest increase in queuing in the fabric RTT for HiPri CliqueMap traffic, albeit with a high baseline latency.

Repeating the same experiment using 1RMA as a transport, we can see that 1RMA on Aquila offers a much lower baseline (less than 5 μs median and tail latency) despite sharing the same GNet fabric with IP traffic. High priority 1RMA traffic uses different virtual channels than low-priority Pony Express IP traffic and thus is nearly unaffected by adding the bulk traffic (blue and red bars). Even when low priority 1RMA traffic shares the virtual channels with the bulk IP traffic (yellow and green bars), the overall latency is slightly higher than 10 μs but still lower than Pony Express traffic on Ethernet networks (orange and cyan bars).

Effect of Burst Size. One of the lessons we learned in Aquila is the phenomenon of *self-congestion*. The IP network in Aquila has an injection rate of 100 Gbps per TiN while

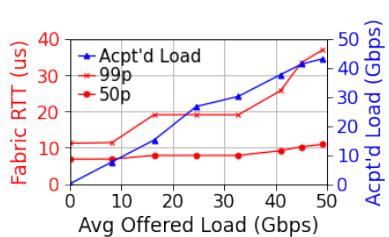


Figure 7: IP Latency vs. Load: Fabric queuing remains low under load.

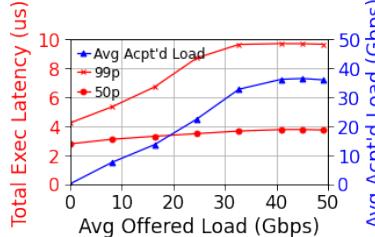


Figure 8: RMA read latency under varied RMA Load.

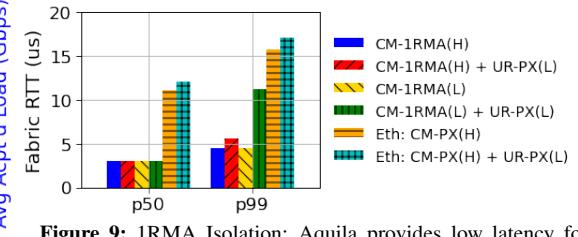


Figure 9: 1RMA Isolation: Aquila provides low latency for 1RMA traffic, even when sharing the network with IP (H = High Priority, L = Low Priority, CM = CliqueMap, PX = Pony Express)

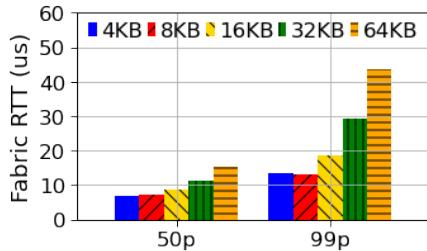
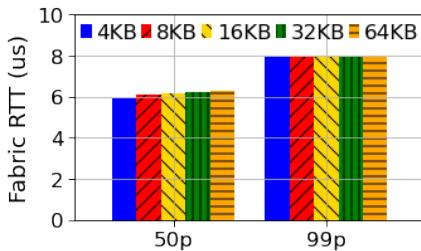


Figure 10: Effect of burstiness on queuing in a full sized Aquila (left) and a half sized Aquila (right). By keeping the injection rate constant and varying message size, we can see the effect of burstiness on queuing latency.



the aggregate bandwidth along the minimal paths between two pods in the full scale Aquila topology is limited to 50 Gbps. This leads to cells taking non-minimal routes even if the overall injection rate is well below the link rate due to bursts. We show this effect by keeping the injection rate of a point-to-point traffic at 0.6 Gbps but varying the message size of the RPC using Pony Express. Varying the message size only affects the burstiness of the injection. Figure 10 shows that as we increase message size, the tail fabric latency increases past 40 μ s. However, repeating the same experiment in a half-scale version of Aquila where we have matching inter-pod bandwidth to the IP injection rate from each TiN, we see no effect of message size on queuing in the fabric. Provisioning higher minimal path bandwidth trades off better performance under bursty traffic conditions in exchange for a smaller maximum scale of the topology.

5.2 Application Impact

In order to see application impact, we compare CliqueMap lookup (of objects with 4 KB size) latency using 1RMA and Pony Express as a transport for RMAs on the Aquila network. We use O(100) backends and clients and vary queries per-second from each client. Figure 11 shows how 1RMA on Aquila cuts the median and tail latency by 50% at low QPS and by more than 300% at high QPS. As with prior work [51], higher load levels with 1RMA deliver *lower* latencies, as individual servers may dwell in low-power states at low load.

6 DISCUSSION

While our approach to Aquila’s design enabled us to develop a unified low latency network fabric for datacenter networks, there were a number of challenges that we had to overcome. We highlight some of the key challenges next.

Single chip part and direct connect topology. While the single chip design delivered a sustainable development model

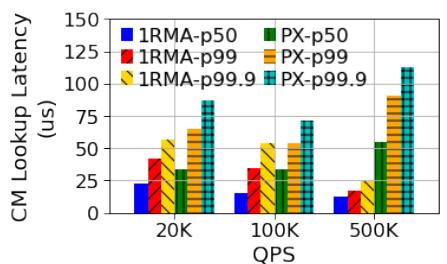


Figure 11: Effect of a cell-based RMA protocol on end-to-end CliqueMap lookup latency.

with a modest sized team and cost efficiency for the Aquila network, the approach had a couple of key implications on the architecture and deployment. First, the single part implied that we had to deploy Aquila as a direct connect network topology because an indirect topology (such as a Clos network) was infeasible with TiN chips. While not a drawback by itself, a direct connect topology is not conducive to incremental deployment. Secondly, the evolution of the NIC and the switch architectures were coupled together from a multi-generational roadmap standpoint.

For simplicity, we designed the Aquila Clique as a homogeneous unit of deployment without an intent to mix hardware from different generations. Moreover, the networking footprint for the entire Clique (up to 24 racks housing all TiN cards as well as the networking fiber) was designed to be deployed up front and host machines could be incrementally populated on demand. With an indirect topology, a small number of network racks (e.g., 4) could be pre-deployed with server racks deployed incrementally. With a direct topology, all server racks (potentially without servers) had to be pre-deployed. Further, for a given optical technology, an indirect topology supports more deployment flexibility: all server racks need only be within a (say) 100m radius of the network racks. With our direct topology, care had to be taken to lay out the rack footprint such that the GNet fiber length between all rack pairs was within the budget of (say) 100m.

Self congestion due to thin minimal path links. The scale of a Dragonfly topology can be increased until we have only a single global link between each pair of pods. However, a mismatch between the injection bandwidth from a TiN and the pod-to-pod bandwidth leads to *self-congestion* where, even at low loads and especially for large MTU packets, some cells may be routed minimally while others may traverse non-minimal paths. As a result, there is some vari-

ance in latency introduced due to cell and packet reassembly even for point-to-point flows at low average loads.

For our initial Aquila prototype, we chose a Clique size of 576 TiNs where the pod-to-pod bandwidth was 2x25Gbps which was 1/4 the maximum injection bandwidth of 200Gbps for each TiN, a balance between Clique scale and self-congestion in the Dragonfly configuration. Further, we tuned adaptive routing to switch from minimal to non-minimal paths to reduce the variance due to self-congestion.

Overhead of cell switching and solicitation. Cell switching and solicitation are key features in Aquila for achieving predictable, low network latency. Switching GNet cells comes with an overhead of approximately 5% due to an 8 byte GNet header for each 160 byte GNet cell. The RTS/CTS solicitation for each IP packet incurs a latency overhead of an extra round trip through the network though the RTS/CTS cells get high priority through the GNet network and the overhead is further mitigated for packets with large MTU. We considered both these overheads acceptable in exchange for low tail latency even at high injected loads. Considering a larger GNet cell size as well as the ability to not incur solicitation overhead at low loads are techniques we are investigating to further mitigate these overheads.

Debugging a cell switched network. Since the Aquila Clique is not an IP routed fabric internally, standard debug tools such as traceroute only show 1 hop through the entire Aquila fabric. To debug data blackholes in Aquila, we implemented a *cell tracing* capability in TiN. Cells that are marked with a bit are sampled by each TiN in the cell’s path and sent to a central collector over UDP. The collector can then stitch the path of the constituent cells of a packet and triangulate any mis-configured or faulty hardware.

Limited RAM on TiN and low level firmware API. To save cost and board space, we provisioned just 2MB of RAM for the firmware running on the TiN chip, which led us to a custom firmware implementation. Firmware development added significantly to the development effort, since many basic facilities had to be customized or re-implemented (e.g., logging, memory allocation, and flash storage).

The decision to expose a register level API to the SDN controller for programming the TiN chip had the benefit of shifting complexity away from the resource constrained firmware as well as simplifying the capability to upgrade firmware with Non-Stop Forwarding (NSF). It also meant that new features could be implemented without needing to roll out a new firmware version since all features of the hardware were exposed. A challenge with this approach was maintaining this interface across multiple hardware generations, since the SDN controller would need to be aware of the register level details of each chip.

For future designs, we are investigating adding more compute to the NIC so that it can be Linux based. Adding RaspberryPi equivalent compute to each NIC is likely to minimally increase the per unit cost relative to the expected gains in de-

velopment velocity. Additionally, more compute will unblock the use of an API with a higher level of abstraction, such as P4 Runtime [24].

Legacy Applications Performance. While Aquila delivered significant application performance improvements (§5) for the co-designed case, such as CliqueMap with 1RMA, it did not have a significant positive impact on legacy applications. We observed that legacy application’s tail latency is dominated by the host software stack, including thread wake up latency. Moreover, with IP software stacks, RTS queue length is governed by the host congestion control algorithms rather than the GNet fabric cell latency. Looking forward, we are shifting transport and network protocols to natively take advantage of future-looking hardware improvements, creating an interesting tension where the substantial software investment would likely not be a net positive until newly designed hardware is deployed across the majority of the fleet.

7 RELATED WORK

Topology and Cell-switching. Aquila uses a direct-connect topology, Dragonfly [30]. The Cray Cascade system [17] utilizes a Dragonfly topology as the basis for an HPC fabric. This design uses a high radix switch with 4 integrated host interfaces, using a proprietary packet format and virtual cut-through. The gateway to Ethernet networking requires processing nodes connected to both types of network. Aquila differs from this system (and from work on Flattened Butterflies [31] and HyperX [3]) by using the topology as a cell fabric, as opposed to a packet network with virtual cut-through. JellyFish [52] is a random-graph topology with its own challenges of deployment. Sirius [6] is a flat-topology with similar goals to Aquila but it utilizes optical circuit switching rather than cell or packet switching. Early ATM networks provided Ethernet-on-ATM [26]. More recently, Stardust [56] employed the idea of cells to give a higher effective switch radix by using single lane channels in the fabric.

Low latency networking protocols. Infiniband [10] implements an alternative networking stack to Ethernet/IP optimized for lower latency. This provides a flow controlled, lossless packet level protocol, a reliable transport implementation, and a complete set of messaging and RDMA operations. Although inter-operation with Ethernet networks for IP traffic can be implemented by gateway functions, Infiniband is commonly used as a dedicated HPC network. SRD [47] (Scalable Reliable Datagram) is an alternate transport protocol layered over IP datagrams that is used in conjunction with EFA (Elastic Fabric Adapter) by a Cloud computing provider to provide lower latency communications services for HPC applications. This has the advantage of being able to use standard Ethernet switches at some cost in minimum achievable latency. While Aquila uses cell-based adaptive routing, SRD uses source-based adaptive multi-pathing.

Congestion control. Solicitation is one of the key elements of GNet for controlling congestion in the GNet fabric. A

few recent congestion control schemes such as Homa [42], NDP [25], ExpressPass [11], pHost [19] and Stardust [56] use a receiver-driven solicitation scheme, similar to that of GNet, to avoid incast congestion and achieve low latency. Aquila’s solicitation controls the transfer of IP packets from buffers at the GNet fabric edge and does not directly control the IP NIC. This means it can handle both gateway and host interface IP traffic, but it requires host-based congestion control [33] to cause the traffic sources to back off in the event of congestion.

Control-plane. Aquila’s control plane was designed with a distributed software defined control-plane. Most of the previous SDN controllers, Onix [32], ONOS [9], Flowlog [43], Ravel [54] assume routing of IP traffic and rely on OpenFlow to program switches. Aquila’s control plane introduces a lower-level communication protocol from a Switch Front-end module to control light embedded switch controllers. The table-based design in [18, 43, 54] allowed for extending routing and sequencing to support GNet flows in addition to IP flows.

8 CONCLUSION

In this paper we present Aquila, our first foray into tightly-coupled networks (Cliques) integrated within the datacenter networking ecosystem realizing Clique-scale resource disaggregation and predictable, low-latency communication. Our primary goal is to advocate for a new design architecture for datacenter networking around Cliques and to encourage new research and development in tightly-coupled networking in support of high-performance computing, ML training, and network disaggregation while simultaneously interoperating with traditional TCP/IP/Ethernet traffic at datacenter scale. We believe our experience, both positive and negative, with the Aquila prototype will set the foundation for future exploration in this space.

Acknowledgments We would like to thank David Culler, John Wilkes, David Wetherall, the anonymous NSDI reviewers and our shepherd, Brent Stephens, for providing valuable feedback. Aquila was a multi-year effort at Google that benefited from an ecosystem of support and innovation. Many contributed to the work, including but not limited to Adam Jesionowski, Alan Lam, Alex Smirnov, Amir Salek, Brandon Ripley, Chip Killian, Daniel Nelson, David Wickeraad, Deepak Arulkannan, Deepak Lall, Duncan Tate, Jakov Seizovic, Jeffery Seibert, Jennie Hughes, Joe Love, Kamran Torabi, Luiz Mendes, Matt Maxwell, Matthew Beaumont-Gay, Philippe Selo, Phillip La, Ranjan Bonthala, Robin Zhang, Scott Berkman, Sean Clark, Shaun Tran, Simon Sabato, Steven Knight, Trevor Switkowski, Tri Nguyen, Warren James, Wilson Lee, Yousuf Haider, and Zhenchuan Pang.

REFERENCES

- [1] Freertos: Real-time operating system for microcontrollers. <https://www.freertos.org/>. Accessed: 2022-02-28.
- [2] Lwip: A lightweight tcp/ip stack. <https://savannah.nongnu.org/projects/lwip/>. Accessed: 2022-02-28.
- [3] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S Schreiber. Hyperx: topology, routing, and packaging of efficient large-scale networks. In *2009 SC Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2009.
- [4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.
- [5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS ’12, pages 53–64, New York, NY, USA, 2012. ACM.
- [6] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM ’20, page 782–797, New York, NY, USA, 2020. Association for Computing Machinery.
- [7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.
- [8] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don’t mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.
- [9] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O’Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6, 2014.

- [10] Rajkumar Buyya, Toni Cortes, and Hai Jin. *An Introduction to the InfiniBand Architecture*, pages 616–632. 2002.
- [11] Inho Cho, Keon Jang, and Dongsu Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the ACM SIGCOMM 2017 Conference, SIGCOMM ’17*, pages 239–252, New York, NY, USA, 2017. ACM.
- [12] David D. Clark, John Wroclawski, Karen R. Sollins, and Robert Braden. Tussle in cyberspace: Defining tomorrow’s internet. *IEEE/ACM Trans. Netw.*, 13(3):462–475, June 2005.
- [13] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2c2: A network stack for rack-scale computers. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 551–564, 2015.
- [14] William J Dally. Virtual-channel flow control. *ACM SIGARCH Computer Architecture News*, 18(2SI):60–68, 1990.
- [15] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.
- [16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.
- [17] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray cascade: A scalable hpc system based on a dragonfly network. In *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–9, 2012.
- [18] Andrew Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google’s software-defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.
- [19] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT ’15*, pages 1:1–1:12, New York, NY, USA, 2015. ACM.
- [20] Marina García, Enrique Vallejo, Ramón Beivide, Miguel Odriozola, and Mateo Valero. Efficient routing mechanisms for dragonfly networks. In *2013 42nd International Conference on Parallel Processing*, pages 582–592. IEEE, 2013.
- [21] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM ’16*, page 58–72, New York, NY, USA, 2016. Association for Computing Machinery.
- [22] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsy, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi. Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10, 2016.
- [23] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VI2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication, SIGCOMM ’09*, page 51–62, New York, NY, USA, 2009. Association for Computing Machinery.
- [24] The P4.org API Working Group. P4 Runtime Specification. <https://p4.org/p4runtime/spec/v1.2.0/P4Runtime-Spec.html>, 2020.
- [25] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichik, and Marcin Mojcik. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the ACM SIGCOMM 2017 Conference, SIGCOMM ’17*, pages 29–42, New York, NY, USA, 2017. ACM.
- [26] Hong Linh Truong, W. W. Ellington, J. Y. Le Boudec, A. X. Meier, and J. W. Pace. Lan emulation on an atm network. *IEEE Communications Magazine*, 33(5):70–85, 1995.
- [27] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hözle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In

- Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM ’13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.
- [28] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gotipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA ’17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.
 - [29] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, April 2012. USENIX Association.
 - [30] John Kim, William J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, pages 77–88. IEEE, 2008.
 - [31] John Kim, William J Dally, and Dennis Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 126–137, 2007.
 - [32] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.
 - [33] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM ’20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.
 - [34] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony Rowstron, Hugh Williams, and Xiaohan Zhao. Xfabric: A reconfigurable in-rack network for rack-scale computers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 15–29, 2016.
 - [35] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1171–1186, 2020.
 - [36] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, Lombard, IL, April 2013. USENIX Association.
 - [37] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP ’19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.
 - [38] Nick McKeown. The islip scheduling algorithm for input-queued switches. *IEEE/ACM transactions on networking*, 7(2):188–201, 1999.
 - [39] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. 38:69–74, 2008.
 - [40] Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing*, pages 255–312. Kluwer, 2000.

- [41] 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 USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 403–418, 2020.
- [42] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM ’18*, pages 221–235, New York, NY, USA, 2018. ACM.
- [43] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 519–531, 2014.
- [44] Open Networking Foundation. Mission of open networking foundation. <https://opennetworking.org/mission/>, 2021. Accessed: 2021-03-08.
- [45] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network’s (Datacenter) Network. In *Proceedings of the ACM SIGCOMM 2015 Conference, SIGCOMM ’15*, pages 123–137, New York, NY, USA, 2015. ACM.
- [46] Amedeo Sapienza, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. April 2021.
- [47] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 40(6):67–73, 2020.
- [48] Arjun Singh. *Load-balanced routing in interconnection networks*. PhD thesis, Stanford University, 2005.
- [49] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagal, Hanying Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network. In *SIGCOMM ’15*, 2015.
- [50] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo MK Martin, Amanda Strominger, Thomas F Wenisch, and Amin Vahdat. Cliquemap: productionizing an rma-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 93–105, 2021.
- [51] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 708–721, 2020.
- [52] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, 2012.
- [53] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.
- [54] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. Ravel: A database-defined network. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2016.
- [55] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *Proceedings of the Ninth European Conference on Computer Systems*, page Article No. 5, 2014.
- [56] Noa Zilberman, Gabi Bracha, and Golan Schzukin. Stardust: Divide and conquer in the data center network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 141–160, 2019.

A HARDWARE PACKAGING DETAILS

Incremental deployment is a much more significant consideration for datacenter systems than for supercomputers which are typically installed as a single system, or in a number of predefined phases. Incremental network deployment is challenging for the Dragonfly topology, where growing the size of the fabric requires the topology to be reconfigured to fully exploit the available chip bandwidth. To avoid recabling for expansion, which is hard to reconcile with the availability requirements of a datacenter, we developed a packaging strategy that allows all the networking infrastructure to be landed as one initial deployment, with the servers being populated incrementally as required.

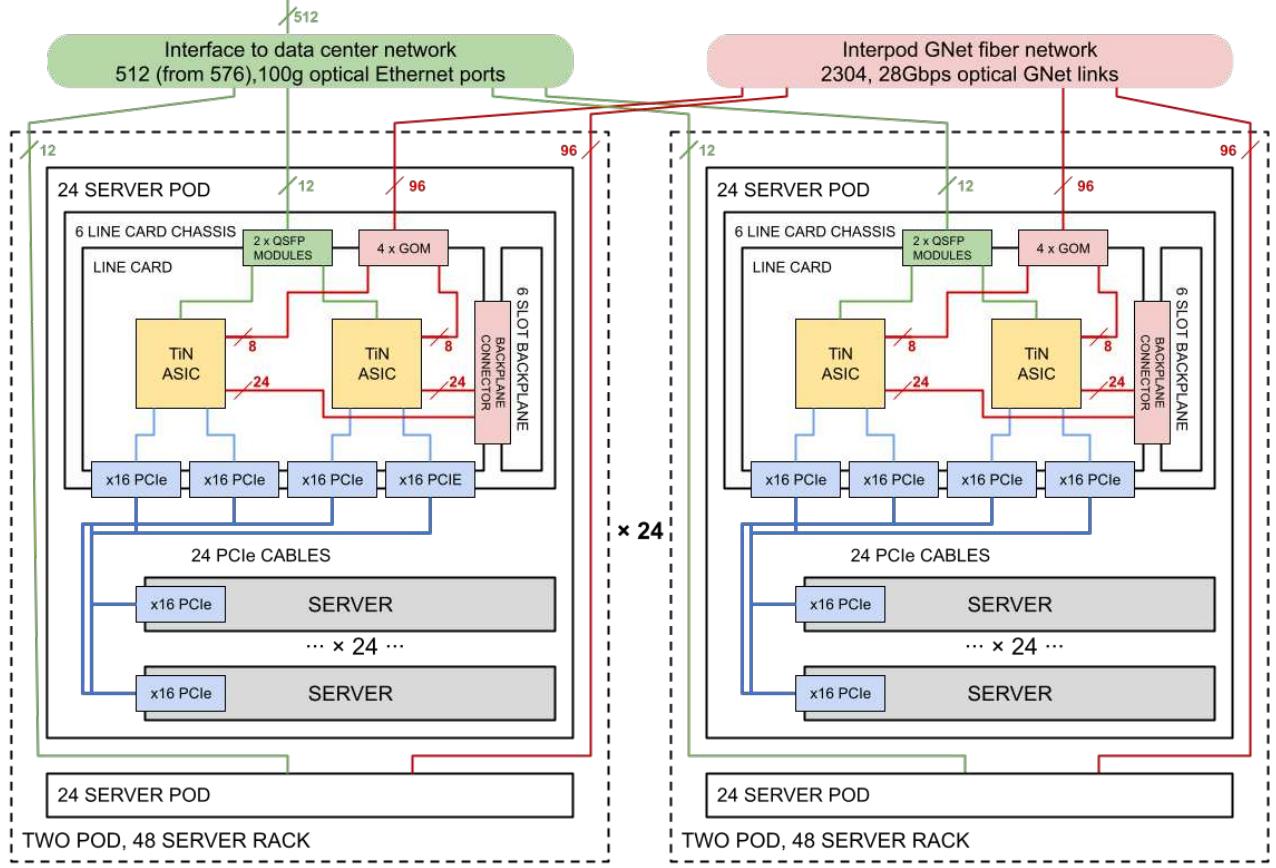


Figure 12: Aquila Clique with 1152 servers, in 24 racks.

A second key consideration was whether to use a blade based system, with a combined server and networking packaging solution, or to work with our existing servers designed around conventional NICs. The latter approach was chosen to avoid having to support two packaging variants of each different server type. These two decisions broadly determined our packaging design.

The physical design (Figure 12) supports up to 48 machines per rack, organized as two pods of 24 servers. The Aquila networking for each pod is provided by a switch chassis containing 12 TiN ASICs, on 6 line cards. The first level interconnect of the Dragonfly is implemented in copper on the switch chassis backplane. Servers are connected to the switch chassis using a cabled x16 Gen 3 PCIe bus. Sideband signals on the cable carry the independent machine management interface from the TiN chip that connects to the server's NC-SI port.

The overall Aquila Clique consists of 24 racks. The connectivity between the racks is optical using custom low cost VCSEL based 4 channel GNet optical modules, 4 per line card. This gives a total of 96 optical GNet connections for the global interconnect level of the Dragonfly from each pod. As there are a total of 48 pods in a clique there are two optical GNet global links between any pod pair. If we connected these directly with two channel fiber ribbons this would re-

quire $47 \times 48 / 2 = 1128$ unique interpod cables to be connected. To simplify the rack to rack cabling we use fiber shuffles within groups of 4 pods to consolidate into wider fiber ribbons allowing the use of 8 GNet link, MPO16 fiber cables. This reduces the rack to rack cabling to 66 4-cable bundles running between 12 pairs of racks greatly simplifying the fiber deployment.

The total number of available 100g Ethernet ports available for connection to the data center spine network from the TiN ASICs is 576. 24 of these are used for rack management. Either 256 or 512 ports are connected to the higher level Ethernet fabric with the remaining 40 ports unused.

A.1 Failure Domains

A key consideration of the Aquila architecture was to reduce the blast radius of any networking component failure. In a conventional network the loss of a TOR impacts all the attached servers; this could be as many as 48 machines for a high radix switch device. In contrast, with the Aquila architecture, loss of a TiN ASIC impacts a maximum of two servers. In practice because the physical packaging solution uses a pair of TiN ASICs on a single line card, the effective blast radius for a repair operation can be up to four servers if 2 servers share a TiN. A switch chassis failure impacts a maximum of 24 servers, however the only chassis components with a significant failure rate are the fans, and N+2 fan redundancy

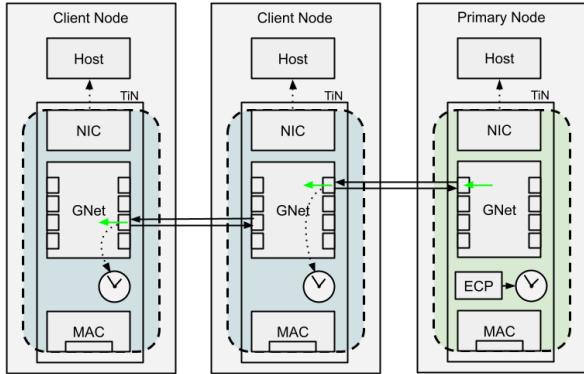


Figure 13: Aquila Clock Sync.

is implemented to minimize the possibility of a chassis level failure.

B CLOCK SYNCHRONIZATION

B.1 Overview

The timesync protocol on Aquila was designed with the aim of keeping the software overhead for timesync low while also providing a tight bound on the notion of current time across the TiNs in the clique. This protocol maintains a single primary clock in the clique against which clients are synchronized purely in hardware (Figure 13). Synchronization is carried out over the GNet links on the switch side of the TiN by transmitting information as lightweight, 8-Byte “ordered sets” between cells, a class of which (TimeSync) are defined for Clique time synchronization. Clients in the host, expecting an IEEE 1588-like protocol to maintain time in the NIC, are able to query the value of this clock. The hardware also corrects for link delays between neighboring TiNs and for time spent within the chip while waiting for a gap between cells to get on the link.

B.2 Implementation

The Timesync hardware on TiN maintains the current time by counting cycles of the core clock along with status bits which tracks several parameters that indicate the accuracy of

the clock. The value of time is also updated by the reception of timesync messages from the neighboring TiN if the current TiN has been configured to be a client node in the time distribution network. The protocol relies on software to set up this time distribution tree [35].

Once the time distribution tree is configured, the TiN transmits TimeSync ordered sets on a configured number of output GNet links at a fixed interval (typically, about 100us). On a client node, an incoming TimeSync message also causes an update to be sent downstream even if the configured interval between messages has not expired. This is to ensure that even the farthest nodes in the time distribution tree do not drift much from the primary node.

The TimeSync message cannot interrupt a cell on the wire, so the ordered set can wait up to 128ns to get onto the wire. Regardless of the delay, the ordered set indicates the actual time of transmission (+/- 2.5ns) by incrementing the value of current time in the TimeSync message for each cycle that it waits to get onto the wire, including flight time across the chip from the hardware clock, arbitration time to get onto the wire, etc. Each receiving TiN is configured to receive TimeSync messages only on a single port and it adjusts for any on-chip delays to get the TimeSync message to the hardware clock along with the delay through the GNet channel.

The delay through the GNet channel is configured on the receiver by running round trip delay measurement at the time of setting up of the time distribution tree. This is done by the GNet ports by putting them in a “latency measurement” mode where the neighbors exchange special ordered sets and reflect the delay through the channel to software as a status.

On reception of a Timesync message, the client node, checks the validity of the message through comparison of status bits transmitted with the message and the difference between the incoming time and the current time against a configurable threshold. The update to current time is only applied when valid Timesync messages are received and if enough invalid messages are seen, the client node signals that a failure is detected. The protocol relies on software to take action once failure is detected

RDC: Energy-Efficient Data Center Network Congestion Relief with Topological Reconfigurability at the Edge

Weitao Wang[†], Dingming Wu^{*}, Sushovan Das[†], Afsaneh Rahbar[†], Ang Chen[†], and T. S. Eugene Ng[†]

[†]Rice University, ^{*}Bytedance Inc.

Abstract

The *rackless data center* (RDC) is a novel network architecture that logically removes the rack boundary of traditional data centers and the inefficiencies that come with it. As modern applications generate more and more inter-rack traffic, the traditional architecture suffers from contention at the core, imbalanced bandwidth utilization across racks, and longer network paths. RDC addresses these limitations by enabling servers to logically move across the rack boundary at runtime. Our design achieves this by inserting circuit switches at the network edge between the ToR switches and the servers, and by reconfiguring the circuits to regroup servers across racks based on the traffic patterns. We have performed extensive evaluations of RDC both in a hardware testbed and packet-level simulations and show that RDC can speed up a 4:1 oversubscribed network by $1.78 \times \sim 3.9 \times$ for realistic applications and more than $10 \times$ in large-scale simulation; furthermore, RDC is up to $2.4 \times$ better in performance per watt than a conventional non-blocking network.

1 Introduction

The importance of the data center network (DCN) has led to a series of DCN architecture proposals [26, 43, 44, 53, 59, 62, 66, 74, 80, 82, 84–86, 96, 110, 118] over the past decade. Although these proposals have competing designs for the network core, the designs for the network edge are similar: servers organized in racks. The network core connects multiple racks, and each rack hosts tens of servers that are connected via a Top-of-Rack (ToR) switch. Standardized racks enable unified power supply and cooling, as well as significant space and cable savings. This rack-based topology and connectivity pattern is deeply ingrained in the design of existing DCN architectures.

While traffic within a rack experiences no congestion, traffic across racks often has to contend for bandwidth due to oversubscription in the network core¹. At the same time, traffic across racks is increasing in data center workloads [36, 37, 40, 99]. Firstly, more and more DCN traffic is escaping the rack boundary due to resource fragmentation [61], large-scale jobs [24], specific application placement constraints for fault tolerance [13], and service-based rack organization for operational convenience [99]—e.g., one rack may host storage servers, and another rack may host cache

¹The literature suggests that there exists a wide-range of common over-subscription ratios between 4:1 to 20:1 [43, 62, 99, 105].

servers. Secondly, there is also an increasing amount of traffic that leaves the pod. For instance, a web-frontend cluster may need to retrieve data from a database cluster or submit jobs to a Hadoop cluster [99].

Thus, the need for efficient handling of cross-rack traffic has motivated numerous approaches; but they have one thing in common – they view the rack design (i.e., a ToR switch connecting tens of servers) as a given. Firstly, the non-blocking network and its alternatives [26, 62, 64, 65, 82, 109] aim to enlarge the capacity of the network core. However, due to the scaling limit of CMOS-based electrical packet switches [6, 33, 34, 49, 50, 57, 91, 104, 105], building such a network while staying within the datacenter power budget is challenging [107]. Secondly, rack-level reconfigurable networks [53, 74, 80, 110, 118] add additional bandwidth between the most intensively communicating racks with extra cables, lasers, or antennas to relieve the bottleneck at the core. However, the performance improvement is constrained by the fact that the number of additional paths is usually limited. Thirdly, smarter job placement and execution strategies [39, 40, 45, 46, 71, 72, 87, 108, 116, 120] can also reduce the inter-rack traffic by arranging the jobs based on their traffic pattern. However, these placement solutions cannot perform well if traffic patterns fluctuate at runtime or if the application dictates placement and forces the traffic to be cross-rack.

This paper studies a complementary and little-explored point in the design space, which we call the *rackless data center* (RDC) architecture. It logically removes the fixed, topological rack boundaries while preserving the benefits of rack-based designs, e.g., organized power supply and cooling, and space efficiency. In RDC, servers are still mounted on physical racks, but they are not bound statically to any ToR switch. Rather, they can move logically from one ToR to another. Under the hood, this is achieved by the use of the circuit switches (CS), which can be dynamically reconfigured to form different connectivity patterns. In other words, servers remain immobile, but circuit changes may shift them to different topological locations. Therefore, this new architecture is not committed to any static configuration, so servers that heavily communicate with each other can be grouped on demand, and they can be regrouped as soon as the pattern changes again. Such dynamic server regrouping enabled by RDC leads to performance benefits in many common, real-world scenarios (details in §2).

We make the following contributions: 1) a novel architecture called RDC, which can be reconfigured to connect servers under different racks in the same logical locality group despite physical rack boundaries; 2) a low-latency RDC control plane and algorithms, which continuously optimize the RDC topology based on the traffic patterns; 3) a prototype of RDC in both testbed and simulation settings, demonstrating that RDC boosts the performance of a 4:1 oversubscribed network by $1.78 \times \sim 3.9 \times$ for realistic applications and more than $10 \times$ in large-scale simulation; furthermore, RDC is up to $2.4 \times$ better in performance per watt than a conventional non-blocking network.

2 Motivation

RDC is motivated by inefficiencies that stem from the inherent rack boundaries in today’s data centers. RDC enables dynamic topological reconfiguration to regroup servers, leading to improved performance for modern workloads. We propose to realize RDC using circuit switching technologies.

2.1 Rack sizes are inherently limited

Today’s DCNs are organized in physical racks as the basic unit. Communication within a rack is through a ToR switch and enjoys lower latency and higher throughput than that across racks. This rack boundary is stressed by a combination of two trends. First, applications are becoming data-intensive. DNN training, iterative machine learning, HPC, big data frameworks (MapReduce, Spark, HDFS) and many other workloads require extensive data communication. Second, the advent of domain-specific accelerators (GPUs, TPUs) and non-volatile memories (NVM) is further shifting the major bottleneck from computation to network IO. The convergence of these trends leads to the need to maximize rack-level performance as much as possible. Broadcom’s Tomahawk-4 64x400 Gbps—the fastest Ethernet switch ASIC commercially available on the market today [7]—only supports a rack boundary of tens of servers while maintaining maximum rack-level performance. A few years ago, the End-of-Row architecture was proposed as an alternative, where multiple racks of low port speed servers were connected to a high-radix edge switch to form a larger logical rack [1]. However, high-radix switching is not feasible at high port speeds: 400 Gbps ports are common today, and Ethernet standards are growing to terabit level. Therefore, in the foreseeable future, the physical rack boundaries of tens of servers are here to stay. New solutions are necessary to mitigate inter-rack-level bottlenecks.

2.2 Rack boundaries introduce bottlenecks

1. Jobs fragmented across racks. A job may spread across racks if rack resources are fragmented. This is partly because cluster schedulers assign resources to their own jobs locally [5, 11, 12]; also, dynamic job churns ensure that rack resources aren’t always neatly packed [60, 95]. Such resource fragmentation leads to heavy inter-rack traffic which contends

for bandwidth due to oversubscription.

2. Workloads with dynamic traffic patterns. Many data-intensive applications (e.g., DNN training, HPC) consist of multiple stages, and each stage has a different yet predictable traffic pattern. For example, Distributed Matrix Multiplication (DMM) has broadcast (one-to-many) and shift (one-to-one) traffic patterns among different subsets of servers in every iteration (Fig. 8(e)). When these jobs coexist in a cluster, the overall combined traffic pattern will change dynamically and predictably. For such workloads, no static job allocation is sufficient to localize all the traffic patterns simultaneously.

3. Applications with placement constraints. Applications may intentionally spread their instances across racks to balance load [55] to reduce synchronized power consumption spikes [70], or to achieve fault tolerance [13]. For instance, to increase resilience, some distributed storage systems, like HDFS, require at least one replica to be placed on a different rack. These requirements result in placement constraints that are by design crossing rack boundaries.

4. Imbalanced out-of-pod traffic. In large datacenters, traffic patterns across racks are often skewed, and out-of-pod traffic demand for each rack is different. For example, only 7.3% of the traffic from the frontend servers is inter-pod, comparing to 40.7% for the cache servers [41, 99]. Operationally, data centers tend to group servers based on their types [99]. So, the above heterogeneity of the out-of-pod traffic demand will make some racks’ uplinks highly congested (e.g., cache) while other racks still have unused bandwidth (e.g., frontend).

2.3 Facebook trace analysis: A case study

Methodology. We used a public dataset released by Facebook, which contains packet-level traces collected from their production data centers in a one-day period. The traces were collected from the “frontend”, “database”, and “Hadoop” clusters, sampled at a rate of 1:30 k, and each packet contains information about the source and destination servers [4]. To understand the benefits of removing rack boundaries, we simulate a rackless design by regrouping servers of different racks into “logical” racks using the algorithms presented in §3. We have two major findings.

Observation #1: Intensive inter-rack traffic. The first observation from the traces is that most of the traffic crosses rack boundaries in a pod. Fig. 1(a) shows the heatmap of traffic pattern inside a frontend pod with 74 racks, collected during a 2-minute interval. If a server in rack i sends more traffic to another server in rack j , then the pixel (i, j) in the heatmap will become darker. Intra-rack traffic appears on the diagonal (i.e., $i = j$). The scattered dots show that the traffic does not exhibit rack locality—in fact, 96.26% of the traffic in this heatmap is inter-rack but intra-pod. A similar trend exists for the database trace: 92.89% of traffic is inter-rack but intra-pod. Hadoop trace has more intra-rack traffic but still has 52.49% of traffic being inter-rack but intra-pod.

Implication #1: Regrouping servers improves locality. Fig.

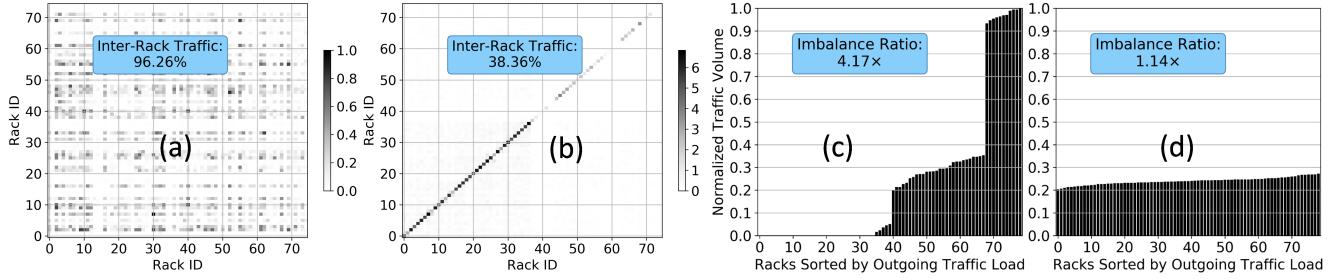


Figure 1: Traffic patterns from the Facebook traces. (a) is the rack-level traffic heatmap of a representative frontend pod. (b) shows the heatmap after regrouping servers in (a). (c) and (d) plot the sorted load of inter-pod traffic across racks in a representative database pod, before and after server regrouping, respectively.

Fig. 1(b) shows the heatmap if servers are regrouped under different racks based on their communication intensity, simulating the desired effect of RDC. Here, most of the traffic is on the diagonal, and inter-rack traffic is reduced significantly to 38.4%. Assuming a 4:1 oversubscribed network, what used to be inter-rack traffic now enjoys 2.82x higher bandwidth. For the database and Hadoop traces, the inter-rack traffic ratios after regrouping are 28.4% and 41.6%, respectively.

Observation #2: Out-of-pod traffic imbalance. Another notable trend is the heavy imbalance of out-of-pod traffic. Fig. 1(c) sorts the racks based on the amount of out-of-pod traffic they sent (traffic trace: database) in a 20-min interval, where the X-axis is the rack ID, and the Y-axis is the (normalized) out-of-pod traffic volume. As we can see, the top 11 racks account for nearly 50% of the out-of-pod traffic, and almost half of the racks never sent traffic across pods. Therefore, some uplinks of ToR switches are heavily utilized, whereas other links are almost always idle. The load imbalance, defined as $\max(L_i)/\text{avg}(L_i)$, where L_i is the amount of out-of-pod traffic from rack i , is as much as 4.17. We found qualitatively similar results on other traces.

Implication #2: Grouping servers mitigates load imbalance. Fig. 1(d) shows the results if servers can be regrouped. In the simulated RDC network, the inter-pod traffic is much more evenly load-balanced across racks, achieving a load imbalance of 1.14. Moreover, the aggregated bandwidth for the out-of-pod traffic increases to 1.79x of the previous bandwidth. This would make better use of the ToR uplinks and avoid congesting any particular link due to imbalance.

2.4 The Power of RDC

Driven by the application-level demand and trace-based analysis, we propose the concept of *rackless data center (RDC)*, which logically removes the physical rack boundaries while maintaining the high-speed rack-level performance. In RDC, servers are mounted on the same “physical rack” sharing the power supply and cooling system but can be logically moved across the ToR switches. We call the new groups of servers served by the same ToR a “logical rack”. Fig. 2 illustrates the benefits of RDC due to server regrouping.

1. Mitigate the effect of resource fragmentation. RDC can

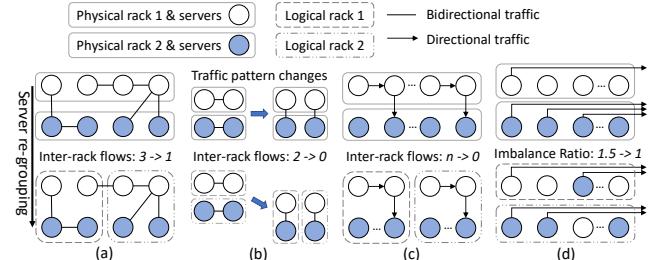


Figure 2: Comparisons between before and after server regrouping for (a) placement optimization, (b) dynamic optimization for evolving patterns, (c) application constraints accommodation, and (d) out-of-pod load balancing.

reduce the effect of resource fragmentation by relocating the heavily communicating server groups under the same logical rack, thus reducing inter-rack traffic. RDC can completely localize smaller jobs that are possible to be packed within one logical rack, like the job on the left-hand side of Fig. 2(a). Even for bigger jobs that cannot be packed within one logical rack, RDC benefits them by (1) localizing as many traffic flows as possible to logical racks, like the job on the right-hand side of Fig. 2(a); and (2) minimizing overall inter-rack traffic from all jobs, leaving the core bandwidth to be shared by much fewer flows that must cross the rack boundaries.

2. Optimize for dynamic traffic patterns. The ability of dynamic server regrouping enabled by RDC can potentially optimize the applications with variable yet predictable traffic patterns. With such changing patterns as shown in Fig. 2(b), RDC is able to dynamically change the topology and minimize the inter-rack traffic for all patterns.

3. Accommodate application placement constraints. As shown in Fig. 2(c), application-level constraints can be accommodated by RDC while localizing traffic. For example, HDFS always requires at least one data block replica to be placed on a different rack. By regrouping the servers from different racks into one logical rack, RDC can place the replicas to a different physical rack but within the same “logical” rack, which provides higher bandwidth and also satisfies the replica placement policy of HDFS.

4. Balance out-of-pod traffic. RDC is able to regroup the servers according to their out-of-pod traffic demands and balance link utilization, hence relieving the bottleneck. In

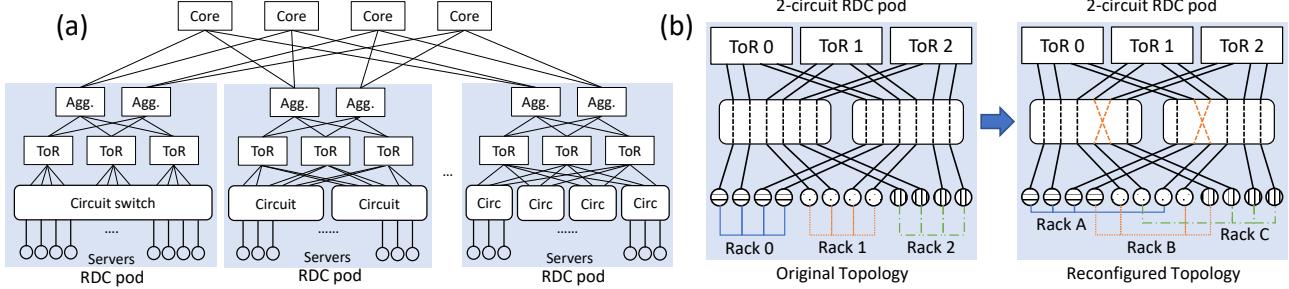


Figure 3: RDC architecture overview. (a) is an example of the RDC network topology. Different numbers of circuit switches can be inserted at the edge between servers and ToR switches. Connectivities for aggregation switches (*agg.*) and core switches remain the same as in traditional Clos networks. (b) shows the original topology and an example reconfigured topology for a 2-CS RDC pod with 3 racks and 4 servers under each rack.

Fig. 2(d), the imbalance ratio has been decreased to 1 from 1.5 after the grouping is changed according to the out-of-pod traffic demand.

2.5 Realizing RDC

Circuit switches (CS) are widely used to provide reconfigurable connections among end points, which is a great fit for the server regrouping functionality of RDC that we discussed above. One realization of RDC is to connect all the servers and all the ToR switches within a pod with a single CS. Alternatively, RDC can also use multiple smaller port count CSes to form a distributed reconfigurable server-to-ToR fabric.

RDC can potentially leverage any kind of CS technologies, including optical and electrical circuit switches alike [104]. However, at high data rates, optical transceivers are the standard interfaces. Therefore, to make the realization long-term sustainable, we consider various optical circuit switching (OCS) technologies. Several OCS technologies are available today such as 3D/2D MEMS, AWGR, etc. Fundamentally, OCS does not perform packet-level processing and forwards the photon beams using mirror rotation, diffraction, etc., which leads to some inherent advantages such as a) agnostic to data-rate (or modulation format), b) negligible power consumption, c) negligible forwarding latency due to no buffering, and d) no need of transceivers at the OCS ports. Additionally, different OCS technologies can provide very fast switching. For example, 2D-MEMS-based OCSes provide microsecond switching [96]), AWGR switches with the latest tunable transceivers can provide nanosecond switching [33, 35, 48, 49, 58, 77]. Moreover, OCS are highly reliable [101] and, due to their simplicity, mostly free from firmware bugs and software misconfigurations.

3 The RDC Architecture

3.1 Connectivity structure

RDC changes the traditional multi-layer Clos topology [26, 62] by inserting one or more circuit switches (CS) at the edge layer between the servers and ToR switches, so that the server can be connected to different ToR switches through circuit reconfiguration. The aggregation and core layers of the net-

work remain the same. Each circuit switch has some ports connected to every ToR switch within the pod to guarantee that the servers could be connected to any ToR switch. For the 1-CS RDC pod, the servers can be grouped without constraints, as long as the number of servers under each ToR switch is the same. If multiple circuit switches are used in one pod, the additional connectivity constraint is that not all the servers under one circuit switch can be connected to the same ToR. With such design, RDC maximizes the flexibility to permute the server-ToR connectivities, allowing the most intensively communicating servers to be localized under the same ToR and enjoy the line rate throughput.

Fig. 3(a) shows an example of the RDC pods. For a pod with m racks and n servers per rack, $2mn$ ports should be provided by all the circuit switches in total to link both servers and ToR switches. For instance, a 16-rack pod with 32 servers can be built with either 1 circuit switch with 1024 ports or k switches with $\frac{1024}{k}$ ports each. Fig. 3(b) gives a detailed example of inserting multiple circuit switches and how to reconfigure for regrouping servers. For a pod with k circuit switches, $\frac{n}{k}$ servers under each ToR are connected to one circuit switch, so that the original topology can keep every server under its own physical ToR switch.

Intuitively, if we increase the number of CSes, the design becomes more distributed which decouples it from a particular CS technology's port count availability; while at the same time, the flexibility of moving servers across the ToRs is slightly reduced. To shed light on this trade-off, we perform trace-based analysis with varying numbers of CSes between the servers and ToR switches. To find a valid server regrouping, we formulate an Integer Linear Programming (ILP) which maximizes the traffic localization given the constraints arising from multiple CSes (more details in §4.3). For the analysis, we consider an RDC pod with 16 ToRs and 32 servers per ToR, having 4 : 1 oversubscription above the ToR level. We vary the number of CS from 1 to 8 and compare the performance with a static 4 : 1 oversubscribed network. Fig. 4 shows a boxplot of the flow completion times (FCT) of these architectures for flow-level Cache traffic trace generated from [99]. We observe that the potential benefit of RDC remains high

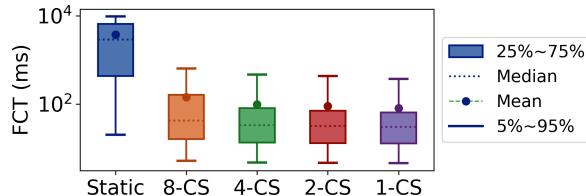


Figure 4: The potential improvement for FCT remains high across a wide range of multi-CS configurations in RDC

across a wide range of CS configurations, which validates the efficacy of our distributed design.

3.2 The RDC Controller

Today’s data centers are constructed from modular pods [3, 14, 19, 22], where a pod typically hosts one type of service. RDC similarly views pods as basic units and uses a per-pod network controller that manages both packet switches and circuit switches within the pod. The controller reconfigures the network at timescales of seconds or longer depending on the traffic pattern. It has two operation modes: it can receive the traffic demands or commands from the applications directly in the proactive mode, or passively monitor the traffic statistics from packet switches in the reactive mode.

We illustrate the workflow for both modes in Fig. 5. The controller 1) first collects the traffic statistics by querying the flow counters on the ToRs, or passively receives the information from the applications; 2) determines the optimized topology with certain optimization goals (§4); 3) generates a set of new routes and pre-installs them on the packet switches; and 4) finally sends the circuit reconfiguration request to the circuit switch and simultaneously activates the new routing rules on packet switches. The first two steps serve as the RDC control plane (discussed later in §4), while the last two steps configure the data plane (discussed in §3.3). Note that only the final step would cause a small amount of disturbance due to the circuit reconfiguration delay.

3.3 Routing

In traditional DCNs, forwarding rules are aggregated based on IP prefixes. In RDC, such aggregation does not work as servers have no fixed locations. Instead, RDC uses per-pod flat IP addressing and exact matching rules on packet switches. Topology changes are captured by updating the routing rules. These rule updates are for intra-pod routing only, as routing mechanisms across pods remain unchanged.

In an RDC pod, each ToR has a flow table entry for every server IP in its rack, and a single default entry for other addresses outside the rack. Each ToR splits traffic to other racks equally across its uplinks using ECMP [69]. All *agg.* switches have the same forwarding table: one entry per destination IP. The flow entries on ToRs and *agg.* switches both need to be updated when topology changes. For ToRs, only the rules for downward traffic need to change; the default ECMP entry for upward traffic remains the same. Therefore, for an RDC pod

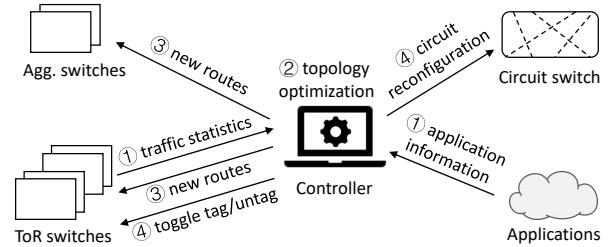


Figure 5: Workflow overview of RDC.

with m racks and n servers per rack, a topology change could result in n rule updates on ToRs and $m \times n$ updates on *agg.* switches, which is on the order of hundreds to a thousand. Updating this number of rules on an OpenFlow switch could take 100ms to over 1s [68, 76]. Previous works have developed the *two-phase commit* method to reduce disruption during updates [81, 98], which first populate the switches with new routing rules and then flip the packet version at the ingress switches. However, such an approach cannot avoid packet loss in the transient state, like changing the packet version rule [81]. This is because updating the packet version rule at the ingress switch requires two rule changes—removing the old rule and installing the new rule—and therefore is not atomic. Our measurement on a Quanta T3048-LY2R OpenFlow switch shows the transient period could last for 0.5ms.

Instead of changing the packet version, in RDC a switch performs binary changes from VLAN tagging packets to not tagging them, and vice versa. The VLAN-tagged packets will match a group of rules with the VLAN tag as a match field, whereas the packets without VLAN tags will match a more general group of rules without VLAN IDs. In this way, adding and removing a single VLAN tag rule achieves the same goal as changing the packet version, but the operation is atomic and avoids packet loss (more details in §A.1.) We apply this update approach to both ToR and *agg.* switches but only tag or not tag packets on ToR switches. Tag flipping actions are only performed when the new forwarding rules have been populated network-wide. The VLAN tag flipping actions need to be executed at the same time across multiple ToRs; such network-wide changes can be performed using well-known SDN time synchronization [90] and consistent update [42, 73, 100] techniques, so that changes can be synchronized and take effect atomically.

3.4 Discussions

Reducing the path length: Besides the throughput benefits mentioned in §2, localizing the hosts under the same logical rack would effectively reduce the average path length (evaluated in §5.2), and thus reduce the network latency. Therefore, low-latency applications and disaggregated systems [56, 63, 92, 94, 103] may benefit from RDC’s design as well.

ToR failure handling: In a traditional data center, a ToR failure disconnects all servers in the rack. ToR failures are handled either by multi-homing servers to several other ToRs

[79, 83] or by replicating applications under multiple ToRs [113, 119]. But in RDC, servers are not tied to any particular ToR, servers under a failed ToR can be migrated to a healthy ToR. To host the relocated servers, we can reserve some “free” ports on each ToR for recovery or install a set of backup ToRs [112, 114]. Specifically, for an RDC pod with m racks and n servers per rack, we only need $\frac{n}{m}$ free ports per ToR (or n ports overall) to recover from any single ToR failure, which incurs a low additional cost and complexity.

CS failure handling: In general, circuit switches are extremely reliable [102]. Commercial OCS products have more than 28.5 years of mean-time-between-failure (MTBF) and come with redundant control processors [2]. However, if a CS failure happens, only a small fraction of servers will be disconnected uniformly under each ToR of RDC, due to its connectivity structure. The most common failure mode for the CS is the power outage. To mitigate this, multiple redundant power supplies can be used for the CS [2]. For further protection, battery backups can be used—since the CS draws only tens of Watts, a battery backup already goes a long way.

4 RDC Control Algorithms

RDC has a general framework to support various topology optimization algorithms, working in two modes to collect the traffic demand matrix and compute the reconfiguration plan.

4.1 Proactive-mode RDC

The *proactive* mode of RDC allows applications to explicitly call the RDC controller via RPC with two APIs: 1) **Traffic demand matrix** can be reported by the applications to request reconfigurations. Along with the demand matrix, RDC controller will request the application to specify one topology optimization algorithm from the algorithms described in §4.3 as well. After receiving the request, the RDC controller will calculate an optimal topology with a specified algorithm and conduct the reconfiguration accordingly. 2) **Raw configuration commands** can also be given directly from the applications. For this method, formatted data to describe the new circuit connections will be sent to the controller, so that the controller could bypass the calculation of the optimal topology and directly used the received configuration to initiate the reconfiguration. An additional benefit for applications to send raw configuration plans is that it enables network-aware job placement and scheduling since the applications know the future network requirements in advance.

There are several scenarios where applications can benefit from telegraphing their intent to the RDC controller: 1) In a case where applications intentionally spread their deployment across racks—e.g., for fault tolerance [40] or for reducing synchronized power consumption spikes [70]—inter-rack traffic patterns are unavoidable in traditional architectures. In RDC, however, such applications can request relevant servers to be grouped together logically. 2) The cluster applications may be allocated with resources from multiple racks due to frag-

mentation. By aggregating those fragmented resources to the same logical rack, RDC improves the bandwidth and reduces the average latency. 3) When applications have changing traffic patterns (e.g., distributed matrix multiplication (DMM) algorithms proceed in iterations with shifting traffic patterns), they can request reconfigurations before the next phase starts to ensure locality throughout the job. 4) Last but not least, RDC could rely on the out-of-pod traffic demands reported by applications to balance the load across different ToR uplinks.

We evaluate three different applications in §5.1 to show the performance of proactive-mode RDC, including HDFS, Memcached, and DMM.

4.2 Reactive-mode RDC

The *reactive* mode of RDC does not require to modify application; it collects traffic statistics from the network in one epoch, and reconfigures the network with an optimized topology for the next, based on the statistics and one of the optimization algorithms from §4.3, specified by the network operators.

Traffic statistics. The RDC controller pulls flow counters from ToRs periodically. A flow counter associates the 5-tuple (13 bytes) of a flow to an 8-byte counter value and thus has 21 bytes in total. Switch memory constraint is traditionally the main concern of maintaining per-flow counters, but this constraint is loosening over the years as the switch SRAM size has been continuously growing. The most recent switch ASICs have 50-100MB of SRAM and can store millions of flow states [18, 88]. As recent DCN measurement works show that the number of concurrent flows per server is on the order of hundreds to a thousand [28, 99], each ToR in RDC would then need tens of thousands of flow counters assuming tens of servers per rack. For instance, assuming an RDC pod with 16 racks and 32 servers per rack, and a counter pulling period of 10s, the control channel bandwidth usage is roughly 8.6Mbps, which is low enough to be feasible.

Demand estimation algorithm. Previous works have shown that data center workloads demonstrate certain degrees of stability [38, 99], and RDC similarly relies on this stability to estimate the traffic demand based on historical data. But the *observed* traffic volumes on ToR switches are biased by the *current* topology, so it is important to estimate the true traffic demand, i.e., the traffic demand when flows are not bottlenecked by the network core. Mitigating such observation bias has been studied in previous work, Hedera [27], and we adopt a similar heuristic.

A flow could be bottlenecked either by the network or by the application itself. We call the first class of flows *elastic* and the second *non-elastic*, and RDC only considers elastic flows. The heuristic is to remove flows from the observed traffic matrix whose sizes are smaller than their fair share. The remaining flows are treated as elastic, and RDC calibrates for potential bias in the counters by computing their idealized bandwidth share (i.e., their bandwidth share if they are only bottlenecked by the host NICs’ capacity) as the es-

src \ dst	0	1	2	3
0	?	?	?	?
1	-		?	?
2	?	?		-
3	?	?	-	

src \ dst	0	1	2	3
0		1/3	1/3	1/3
1	-		1/2	1/2
2	1/2	1/2		-
3	1/2	1/2	-	

Source-side fair share

Destination-side check

Figure 6: Hedera demand estimation example. Each "?" represents one flow from source host to destination, "-" represents no flow between that source-destination pair, and number "1/2" represents 50% of host bandwidth. This example ends in one iteration, but it takes more iterations for a more complicated traffic matrix.

timated demand [27]. Hedera is an algorithm to calculate the max-min fair share rate of each flow within a network. It performs multiple iterations to firstly increase the flow capacities at the source (no greater than the source host capacity) and then decrease the exceeding capacities (sum of enlarged flow capacities subtracting the actual NIC capacity) on each destination host until the flows' capacities converge. A simple demand estimation example that ends with only one iteration is shown in Fig. 6 (More details in §A.3). After convergence, the estimated flow demands are aggregated into a server-to-server traffic matrix for reconfiguration. The effectiveness of this demand estimation algorithm is evaluated in §5.4.

4.3 Topology optimization algorithms

RDC enables a range of topology optimization and reconfiguration algorithms.

1. Traffic localization algorithm reconfigures the network to localize inter-rack traffic, after obtaining the flow demands proactively or reactively. The objective of the localization algorithm is to minimize the traffic demands across the logical racks of the new topology. With this objective, the localization algorithm can be formulated as an Integer Linear Programming (ILP) problem as described in §A.4. However, finding the optimal solution is NP-hard, so we provide heuristic alternatives with balanced graph partition [75] for 1-CS RDC and a simplified algorithm for multi-CS RDC discussed in §A.4. The heuristic algorithms can find a high-quality regrouping plan within tens of milliseconds as shown in Table 2.

2. Uplink load-balancing algorithm spreads out-of-pod traffic across ToR switches for load balancing, relieving the potential congestion on the over-subscribed uplinks. The objective for uplink load balancing (ULB) is to minimize the maximum out-of-pod traffic from one rack. We provide a formal problem formulation and faster heuristic algorithms in §A.2.

3. Mixed optimizations can be developed in RDC to localize the inter-rack traffic and balance the out-of-pod traffic at the same time, e.g., for a *mix* of workloads or applications. To satisfy this goal, the objective of this problem will be minimizing $\alpha T + \beta R$, where T is the total inter-rack traffic demands within a pod, R is the maximum volume of out-of-pod traffic across ToRs, and α and β are the respective weights [40].

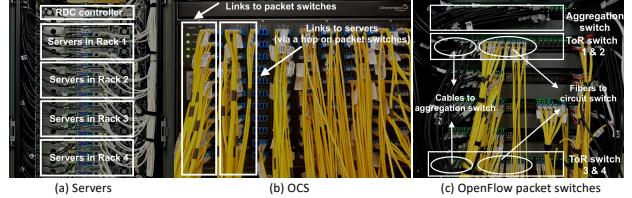


Figure 7: RDC prototype with 4 racks and 16 servers.

4. Scenario-specific optimizations allow applications or network operators to define their own optimization algorithms for regrouping the servers into logical racks. The applications are able to define their own objective function and add more application-specific constraints.

5 Implementation and Evaluation

We conduct comprehensive evaluations using testbed experiments and packet-level simulations. Our experiments focus on several dimensions: a) real-world applications of RDC to HDFS [10], Memcached [23], and MPI-based distributed matrix multiplication (DMM) [54] as use cases, b) packet-level simulations on the latency and throughput improvements at scale, c) packaging, power, and capital cost analysis, and d) microbenchmarks on RDC, including non-disruptive control loop latency.

Testbed. Our RDC prototype consists of 16 servers and 4 ToR switches in 4 logical racks, one agg. switch and one circuit switch; Fig. 7 illustrates our hardware testbed. The ToR switches are emulated on two 48-port Quanta T3048-LY2R switches. Each ToR switch has four downlinks connected to the servers, and one uplink to the agg. switch, forming an over-subscription ratio of 4:1. We can tune this ratio to emulate a non-blocking network by increasing the number of uplinks to 4. The agg. switch is a separate OpenFlow switch. The OCS is a 192-port Glimmerglass 3D-MEMS switch with a switching delay of 8.5 ms. This can also be replaced with other types of OCS. Each server has six 3.5 GHz dual-hyperthreaded CPU cores and 128 GB RAM, running TCP CUBIC on Linux 3.16.5. Most of our experimental results except the large-scale simulation in §5.2 are obtained on this testbed.

Packet-level simulator. In order to simulate a wider variety of experimental settings, we have developed a packet-level simulator based on *htsim*, which was used to evaluate MPTCP [97] and NDP [67]. This simulator has a full implementation of TCP flow control and congestion control algorithms and supports ECMP. We simulate a conservative circuit reconfiguration delay of 8.5 ms, which is what our testbed 3D-MEMS switch achieves. As discussed in §2.5, much faster circuit switching technologies exist [33, 48, 58, 86] that can further improve the performance of RDC. Note that only the circuit that is being reconfigured will experience a disruption; all other circuits continue to function. Packets in flight during reconfiguration will be dropped if they traverse the disrupted links, and unsent packets will be buffered at the servers. We simulate an RDC pod with 512 servers, 32 servers per rack,

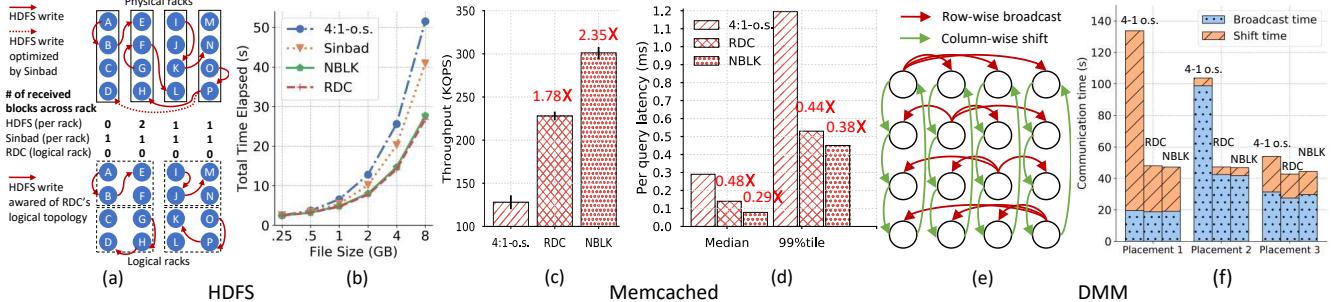


Figure 8: Application performance improvements of RDC compared with the 4:1 oversubscribed network (4:1-o.s.), 4:1-o.s. network powered with Sinbad [45], or the non-blocking network (NBLK). (a) The HDFS write traffic pattern and the number of received blocks per rack. (b) The HDFS transfer time. (c)-(d) Memcached query throughput and latency. (e) The DMM traffic pattern. (f) Average shift time and broadcast time.

and 16 racks overall. The 16 ToR switches are connected to a single agg. switch with tunable oversubscription ratios. Results in §5.2 are obtained via simulation.

5.1 Real-world applications

First, we show how RDC can improve the performance of real-world applications for each of its use cases.

HDFS. We set up an HDFS cluster with 16 *datanodes* across 4 racks and 1 *namenode*, with a replication factor of 3 and a block size of 256 MB. All data blocks are cached in the RAM disk to prevent the hard drive from being the bottleneck. The 16 clients initiated concurrent write requests to 16 HDFS files, respectively. According to the default HDFS data block placement policy, when writing a data block to a datanode, a replica of the block will be placed on the same rack of the original copy, and another replica is placed on a remote rack for resilience (Fig. 8(a)). Therefore, a write operation generates an intra-rack flow and an inter-rack flow.

HDFS can localize all the inter-rack traffic (for storing replicas) by using both proactive RDC and network-aware replica placements. Fig. 8(b) shows the performance gain with RDC and compares it with the non-blocking network (NBLK) and an advanced bandwidth-centric replica placement solution, Sinbad [45]). Sinbad keeps track of the paths and links to reach the replicas within the most recent period and assigns the next replica to the least-utilized paths in the recent period. Therefore, Sinbad does not reduce cross-rack traffic, but can relieve bottlenecks at network links by load balancing as shown in Fig. 8(a). Specifically, it detects traffic imbalance for transferring inter-rack replicas and aims to utilize all links roughly equally—i.e., each rack hosts one replica. In the results, we can see that Sinbad improves the total time for HDFS writes, but still underperforms the NBLK network. In contrast, RDC allows the HDFS to regroup servers directly. Moreover, with the new topology, HDFS could change the replica placement scheme to keep all traffic within the logical racks but satisfy fault-tolerance constraints at the same time, as shown in Fig. 8(a). HDFS with RDC achieves similar performance as the NBLK network, reducing the total time to 0.59× on average, compared to the original topology and

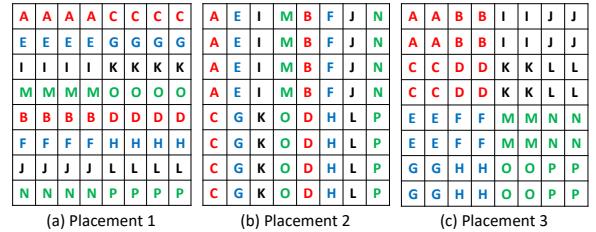


Figure 9: Three different placements for DMM. A-P represent 16 servers and A-D, E-H, I-L, M-P belong to four physical racks separately.

placement policy.

Memcached. We then configured Memcached [23] servers on two racks, and issued read/write requests from two other racks. This emulates the scenario where clients in one pod access cache servers in another pod. Our workload has a) 200 k key-value pairs uniformly distributed across 8 servers, b) a 99%/1% read/write ratio, and c) 512 byte keys and 10 KB values. We adopted a Zipfian query key distribution of skewness 0.99 similar to previous works [31, 93], which led to a load imbalance ratio of ∼1.8 on the server racks.

By reallocating the servers with hot keys equally onto every ToR, RDC improves the query throughput by 1.78× on average and reduces the median latency to 0.48× as shown in Fig. 8(c)-(d). These improvements are close to what a non-blocking network could achieve. RDC also cuts the tail latency significantly, for which network congestion is a major cause [30, 117]. Since in the baseline setting, ToR uplink can easily get congested when several hot keys are coincidentally located in the same rack, even if the overall uplink utilization is low. In contrast, RDC can observe the traffic patterns due to the hot keys, and spread the servers hosting these keys to different racks. This reduces the peak uplink utilization.

OpenMPI DMM. We set up a 16-node OpenMPI cluster across 4 racks and implemented a commonly used DMM algorithm [54] with 64 processes. Matrices are divided into 64 blocks (submatrices). Each server has 4 processes to form an 8 × 8 process layout. Then in each iteration, it performs a “broadcast-shift-multiply” cycle where a process a) broadcasts submatrix row-wise, b) shifts submatrices column-wise, and

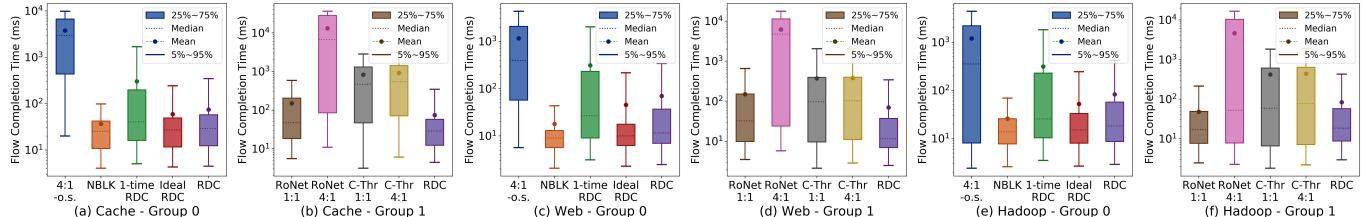


Figure 10: Performance comparison with RDC. Group 0 shows that RDC delivers performance improvements by dynamically reconfiguring the network and achieves similar performance as NBLK; group 1 shows that RDC outperforms alternative designs by benefiting a higher amount of inter-rack traffic.

c) multiplies submatrices as shown in Fig. 8(e). We consider three placements for processes: 1) Fig. 9(a): places them row-wise (no cross-rack traffic for broadcast), 2) Fig. 9(b): places them column-wise (no cross-rack traffic for shift) and 3) Fig. 9(c): places them in a mixed manner, considering both broadcast and shift traffic across racks.

By dynamically configuring the topology for different phases during DMM, RDC shrinks the communication time as well as the end-to-end execution time. Fig. 8(f) shows that RDC improves the overall communication time for placements 1, 2, and 3 by $3.9 \times$, $2.3 \times$, and $1.26 \times$ respectively compared to a static 4:1 oversubscribed network, achieving almost the same performance with the NBLK network. Since the applications have evolving traffic patterns, no static process placement is consistently optimal. Out of the three placements, placement 3 jointly minimizes the cross rack traffic for both communication patterns in DMM, outperforming the other two strategies.

5.2 Performance at scale

Next, we evaluate the reactive RDC pods at the data center scale using the packet-level simulator. Our baselines are a) a static non-blocking network (NBLK), b) a static network with 4:1 oversubscription (4:1-o.s.), c) RDC with future traffic-demand information (Ideal RDC), d) a 4:1-o.s. network that applies RDC’s reconfiguration algorithm only once over the entire traffic trace (One-time RDC). e) a hybrid network—like C-Through [110] with 16 4:1/1:1 oversubscribed reconfigurable circuit ToR-pair links in addition to a 4:1-o.s. network, which is similar to Firefly [66], and ProjecToR [59] in terms of performance. f) a novel circuit-core network—RotorNet [86] with 4:1/1:1 oversubscribed ToR uplink bandwidth. Note C-through has the same circuit switching delay as RDC and buffers packets at ToRs during the circuit downtime.

We used the Cache, Web, and Hadoop traffic traces from Facebook. Since the original traces do not contain flow-level information, we generated flow-level traffic based on the sampled packet traces from [99]. Specifically, we inferred the source/destination servers of the flows from the trace, and simulated flow sizes and arrival times based on Figures 6 and 14 in the same Facebook paper. The Cache workload has an average flow size of 680 KB, with 87% being inter-rack. The Web workload has an average size of 63 KB with 96% inter-rack. For the Hadoop workload, the average size is 67.18 KB

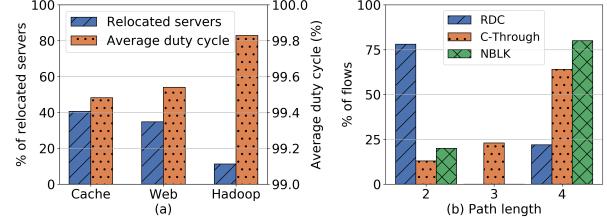


Figure 11: RDC’s average circuit duty cycle is $>99\%$ even with frequent reconfigurations; RDC has an average path length 35% shorter than NBLK.

but only 60% is inter-rack traffic. All traffic traces last for 30s in the simulation, and RDC’s reconfiguration period is 1s.

Fig. 10 shows the boxplot of flow completion times (FCT) for RDC and the baselines using the three traces. We observe that RDC reduces the median FCT by more than an order of magnitude compared to 4:1-o.s. network. Applying RDC’s traffic localization algorithm once can bring some improvements on the median FCT but not as significant as RDC and NBLK, since the traffic pattern changes during the simulation. We found that one root cause for the performance improvements is due to TCP dynamics—severe inter-rack congestion causes consecutive packet losses and TCP becomes very conservative in increasing its sending rate. More importantly, we observe that RDC with future knowledge of traffic demands performs consistently close to the non-blocking network, which again demonstrates the power of a rackless network. Without future knowledge, RDC can still achieve similar performance as NBLK with a slightly longer median FCT, because the cache workload is largely stable at the time scale of seconds, similar to that in the Database workload in the original traces. As for other solutions, C-Through’s average FCTs are at least $3.21 \times$ higher than RDC. Because although C-Through adds extra inter-rack bandwidth, it is provisioned for only 16 ToR pairs. As the traffic traces that motivate our RDC design have more than 16 intensively-communicating ToR pairs (see the heatmap in Fig. 1), C-Through falls short in relieving inter-rack congestion even after enlarging the bandwidths of 16 extra links. 4:1-o.s. RotorNet has the same total uplink bandwidth as RDC, but its performance is much worse than RDC. The non-blocking RotorNet is $2 \times$ and $2.17 \times$ slower than RDC on the Cache and Web traces; only for the Hadoop traces, it can reduce RDC’s average FCT to $0.576 \times$. Since RotorNet provides a dedicated

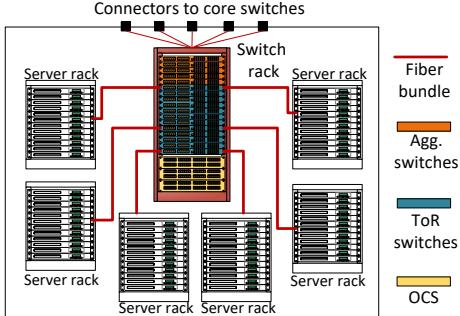


Figure 12: Packaging design of an RDC pod.

link between each ToR pair, if the traffic is skewed between some ToR pairs, it cannot achieve the best performance. So only the Hadoop trace, which has only 60% inter-rack and is quite evenly distributed across different ToRs, enjoys better performance on non-blocking RotorNet.

Fig. 11(a) shows that the average number of servers being relocated in each epoch is different across traces. The duty cycle is an important metric in optical networks to represent the percentage of time that an optical link is up and available for transmission. Assuming one reconfiguration per second, the lowest circuit duty cycle of RDC is 99.2% in theory (details about downtime in §5.4); since not all servers will be relocated in practice, the average circuit duty cycle for all transmissions can be as high as 99.83%. Fig. 11(b) shows the distribution of flow path lengths for RDC, C-Through, and NBLK. (Two C-Through settings have the same distribution; NBLK, 4:1-o.s., and RotorNet also have the same distribution). An intra-rack flow has path length 2 in all networks; and an inter-rack flow has path length 4 in RDC, NBLK, and 4:1-o.s.; the path length could vary in C-Through—3 for the circuit path and 4 for the normal packet-switched path. Overall, RDC localizes more than 70% of the inter-rack traffic and achieves an average path length of $0.75 \times$ of C-Through and $0.65 \times$ of NBLK.

5.3 Packaging, power, and capital cost

Packaging. Fig. 12 shows the packaging design of an RDC pod, which is somewhat different from that of a traditional pod. RDC has a central switch rack dedicated to hosting ToRs, agg. switches, and OCSes. Server racks are connected to OCSes via fiber bundles to reduce wiring complexity. On the central rack, ToRs are connected to OCSes and agg. switches using short fibers and cables, respectively. Agg. switches provide similar connectivity to core switches outside the pod, just like in traditional data centers. To ensure that centralized switch placement has similar reliability as traditional switch placement, backup power supplies are employed. Similar to the existing modular data centers, RDC supports incremental expansion by adding RDC pods.

Power and capital cost modeling. We show that RDC is more economical by comparing the power and capital cost between RDC and NBLK, at 400 Gbps data rate. They both

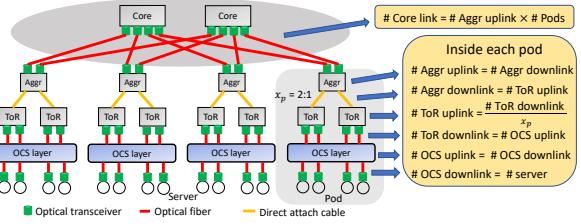


Figure 13: RDC example with detailed components, including the governing equations for power and capital cost model.

Components	Power (Watt)	Cost (USD)	Relative count	
			RDC ($x_p : 1$)	NBLK
Ethernet port [16]	40.6	312.5	$1+4/x_p$	5
Optical transceiver [15]	10	799	$2+2/x_p$	4
Inter-rack fiber [20]	0	6.9	$1+1/x_p$	2
Intra-rack fiber [21]	0	4.9	1	0
DAC [17]	1.5	249	$1/x_p$	1
OCS port [8]	0.14	400 [52]	2	0

Table 1: Power/cost data and relative count of the components for RDC ($x_p : 1$ o.s.) and NBLK at 400 Gbps.

consist of the following types of networking components: a) 400 Gbps Ethernet port, b) 400 Gbps Optical transceiver, c) inter-rack duplex single-mode fiber (average length 10m), d) intra-rack duplex single-mode fiber (average length 3m), e) 400 Gbps Direct Attach Cables (average length 3m) and f) OCS port. NBLK network can use DAC to directly connect the server-ToR downlinks, while RDC needs fiber-optic cables along with optical transceivers both at the server and ToR ends to connect the OCSes in between.

We assume that RDC has an $x_p : 1$ oversubscription above the ToR level. Fig. 13 demonstrates a 4-pod RDC network (total 16 servers) with component-level details (where $x_p = 2$) and shows the governing equations to find the component counts across the network. Based on our modeling, given the number of servers and pods are the same, the relative component count for RDC and NBLK network only depends on x_p . Table 1 shows the recent power and cost values of different components along with their relative count for RDC and NBLK network (400 Gbps). On one hand, the power consumption values of the network components are fundamental and well-documented in datasheets. On the other hand, the component cost can vary based on sales volume, and since we have no proprietary industry pricing figures, we do a "best-effort" calculation based on readily available retail pricing (in other words worst-case or no-discount pricing) for all components, so at least it is somewhat objective and unbiased. We consider \$400 to be the OCS per-port cost, the worst-case price adopted from a recently reported article from Microsoft [52]. Readers should be aware of the limitation of this pricing assumption and take the capital cost results for general guidance only.

As shown in the governing equations in Fig. 13, an $x_p : 1$ RDC network with s servers have s ToR downlink ports, $\frac{s}{x_p}$ ToR uplink ports, $\frac{s}{x_p}$ agg. switch downlink ports, $\frac{s}{x_p}$ agg.

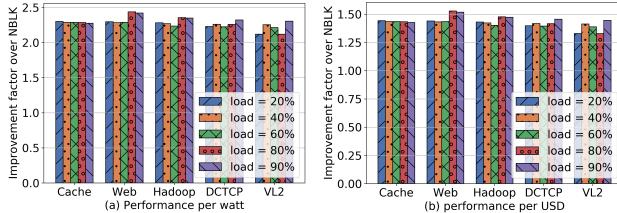


Figure 14: a) RDC (4:1-o.s.) has $2.1\text{--}2.4\times$ improvements in performance per watt than NBLK at 400 Gbps data rate; b) RDC (4:1-o.s.) has $1.3\text{--}1.5\times$ improvements in performance per dollar than NBLK at 400 Gbps data rate, assuming worst-case component pricing.

switch uplinks ports and $\frac{s}{x_p}$ core switch ports; leading to $(s + \frac{4s}{x_p})$ Ethernet ports in total. Consider a traditional NBLK (fat-tree) network with the same number of servers and pods, where the number of Ethernet switch ports at each layer is the same as the number of servers. This leads to a total of $5s$ (using $x_p = 1$ in RDC) Ethernet ports. Hence, the relative Ethernet port count is $(1 + \frac{4}{x_p})$ to 5 (see Table 1). Similar calculation can be applied to other components as well.

Power efficiency. A 4:1-o.s. RDC network consumes $2.29\times$ less power than an NBLK network considering 400 Gbps data rate. RDC significantly improves the performance (median FCT) per watt compared to that of NBLK for diverse traffic patterns across different network loads, as shown in Fig. 14(a). We use five different production traces i.e., Cache [99], Web [99], Hadoop [99], DCTCP [29] and VL2 [62]. For median FCT, RDC has $2.1\times\text{--}2.4\times$ improvements in performance per watt compared to NBLK. RDC also significantly reduces the power consumption of the network because it requires fewer power-hungry packet switches in the core. The optical circuit switch at the RDC edge consumes very little power since it only directs the incoming photon beams using mirror rotation or diffraction.

Capital cost. We again emphasize that readers should take this “best-effort” cost analysis for general guidance only. A 4:1-o.s. RDC network costs $1.4\times$ less than an NBLK network at 400 Gbps. Using the same five production traces, we observe that RDC has $1.3\times\text{--}1.5\times$ improvements in performance (median FCT) per dollar compared to NBLK, as shown in Fig. 14(b). We also estimate OCS per-port cost which would let 4:1-o.s. RDC has an equal performance per dollar as NBLK: it ranges from \$1000-\$1300.

5.4 RDC reconfigurations

To have a deeper understanding of RDC, we break down this analysis into the effectiveness study of the demand estimation algorithm, the non-disruptive control loop before the reconfiguration, and the hardware transient state during the reconfiguration.

Effectiveness of demand estimation algorithm To show the effectiveness of our demand estimation algorithm, we examine how our heuristic interacts with consecutive topology

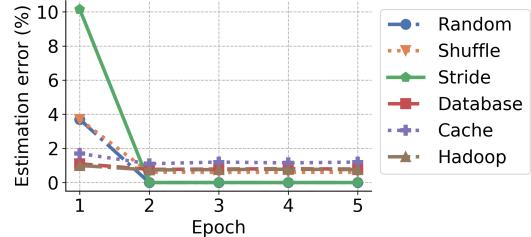


Figure 15: Average demand estimation error over multiple consecutive epochs (epoch duration: 10s).

reconfigurations, by an in-depth study of traffic localization.

We use the same packet-level simulation as §5.2 to illustrate this demand estimation technique. For each simulation, RDC performs traffic localization and reconfigures the topology once every 10s according to the algorithm detailed in §4.3. The senders and receivers of elastic flows are determined based on a chosen traffic pattern, while non-elastic flows are generated with randomly chosen senders and receivers with a data rate $< 10\text{Mbps}$. The ratio between the number of non-elastic and elastic flows is 10:1. Besides the three trace-derived traffic patterns – Cache, Web, and Hadoop, we also test three synthetic traces as follows. Each traffic pattern remains the same throughout the simulation.

Random: Each host i sends a flow to one of the other hosts with uniform random probability;

Stride: Each host i sends to a set of 31 other hosts with indexes $(i + j * 16)\%num_hosts$, $j \in [1..31]$;

Cache: Each host i sends a flow to another host with index $(i + 32)\%num_hosts$;

Fig. 15 shows the average demand estimation errors over five consecutive reconfiguration epochs for all server pairs. We observe that while the initial demand estimation errors can be moderately high (10%), the errors decrease as the network reconfigures to adapt to the traffic pattern in subsequent epochs. In the first epoch, many elastic flows congest the oversubscribed network core. As a result, their flow counters can be small and they could be misidentified as non-elastic flows. However, as RDC adapts the topology to localize the identified elastic flows, fewer elastic flows are transmitted across racks, congestion in the network core is reduced, and thus more elastic flows are correctly identified. For example, because the elastic flows in the stride pattern are eventually all localized within racks, the demand estimation errors for these elastic flows drop to nearly zero. Therefore, we can see that the Hedera technique is well-suited to RDC—reconfiguring the topology to suit the traffic patterns helps improve the accuracy of demand estimates for the next epoch.

Non-disruptive control loop. Next, we evaluate the latency of the RDC control loop, which includes four components: 1) collecting flow counters, 2) estimating traffic demands, 3) computing new topologies, and 4) modifying forwarding rules. This latency will affect how fast RDC can respond to changing traffic patterns. Note that the reactive RDC uses all four components; for proactive RDC using traffic demand matrix,

#Racks	4		8		16		32	
	TL	ULB	TL	ULB	TL	ULB	TL	ULB
Counter collection	10.6	2.3	21.3	2.6	42.6	3.4	85.1	4.5
Demand estimation	10.8	0.7	24.9	1.1	80.6	1.3	310.6	1.7
Topo. computation	7.8	0.1	28.2	0.1	40.3	0.3	69.3	0.6
Rule installation	32.5	30.6	45.6	30.8	75.6	41.4	147.6	70.6
Proactive - Command	32.5	30.6	45.6	30.8	75.6	41.4	147.6	70.6
Proactive - Demand	40.3	30.7	73.8	30.9	115.9	41.7	216.9	71.2
Reactive	61.7	33.7	120	34.6	239.1	46.4	612.6	77.4

Table 2: Control loop latency breakdown (ms) for traffic localization (TL) and uplink load-balancing (ULB).

only steps 3 & 4 will be executed; for proactive RDC with direct configuration command, only the last step is required.

To obtain these results, we ran a set of experiments using different numbers of racks, with 32 servers per rack, using the traffic patterns from the Facebook traces. The ToR switches are connected to a single *agg.* switch. Since our testbed only has four ToR switches, we emulated more ToR switches using servers and ensured that each server has the same latency for collecting counters and installing routing rules as a physical ToR switch. The number of forwarding rules to be installed is bounded by 32 for the ToR switches and $32 \times \#racks$ for the *agg.* switch. And the number varies depending on the traffic patterns and may be different across switches. The overall rule installation delay is determined by the slowest switch, which has the most number of changes.

Table 2 breaks down the control loop latency for traffic localization (TL) and uplink load-balancing (ULB) use cases. Overall, reactive RDC’s non-disruptive control loop latency before reconfiguration is 612.6ms for TL and 77.4ms for ULB, which are on similar timescales with state-of-the-art traffic engineering techniques [38]. Whereas, proactive RDC can reduce this control loop delay to 147.6 ms and 70.6 ms respectively. Since RDC aims to reconfigure the network at large timescales (e.g., seconds or longer), this control loop is efficient enough to be practical. Note that all the above numbers are obtained with our own testbed. With the cutting-edge high-performance switch hardware [9, 16, 18], the latency can be further reduced to support more frequent reconfiguration.

Reconfiguration transient state. It is important to observe that a circuit reconfiguration in RDC happens only when needed, and for the vast majority of the time, circuits are continuously active. When a reconfiguration happens to a circuit, a transient disruption to that circuit does occur. For example, AWGR and star-coupler-based OCSes are becoming popular as tunable lasers with sub-nanosecond wavelength switching are being fabricated [33, 35, 48, 49, 58, 77]. Considering 400 Gbps link speed and 1 ns of switching delay, only 50 bytes of traffic will be buffered or dropped during the transient phase. Also, 2D-MEMS based OCSes are available, having a reconfiguration delay of few microseconds [96]. Even with a relatively slow OCS in our testbed, our experiments show that RDC provides large performance benefits.

6 Related Work

Various DCN proposals recognize the need for serving dynamic workloads and provision bandwidth on demand with reconfigurable topologies. It can be achieved by adding extra bandwidth to the network by creating ad hoc links at runtime [53, 74, 80, 110, 118], but they mostly focus on providing reconfigurable topology at the rack level, assuming skewed inter-rack traffic. RDC, however, alleviates the reliance on such an assumption and achieves higher performance without adding extra bandwidth. Another line of work constructs an all-connected flexible network core with a high capacity [32, 44, 84–86, 96], but they mostly focus on rack-level rather than edge reconfigurability. Flat-tree [115] is an architecture proposal with partial edge-level reconfigurability, which enables DC-wide reconfigurability by dynamically changing the topology between Clos [26] and random graph [106]. However, the topology modes are limited and only suitable for generally expected workload patterns, e.g., rack-, pod-, or DC-local. Our workshop paper [111] does not contain a detailed design, implementation, or evaluation.

Besides architectural solutions, there are also numerous works that improve flow performance by optimizing task placements. For instance, Sinbad [45] selectively chooses data transfer destinations to avoid network congestion; Shuffle-Watcher [25] attempts to localize the shuffle phase of MapReduce jobs to one or a few racks; Corral [72] jointly places input data and compute to reduce inter-rack traffic for recurring jobs. However, these works all have important drawbacks as they only optimize data transfer for one or two stages of job executions. As we noted before, the traffic pattern may change in different stages of a job’s lifetime. Also, there is a set of research projects that improve network performance at the upper layers in the stack. Optimized transport protocols (e.g., DCTCP [28], MPTCP [97]) and traffic engineering techniques (e.g., Hedera [27], MicroTE [38], Varys [47]) can improve flow performance for many applications.

7 Conclusion

The rackless data center (RDC) is a novel network architecture that logically removes the static rack boundaries, using circuit switching to achieve topological reconfigurability at the edge. In this architecture, servers in different physical racks can be grouped into the same locality group at runtime based on traffic patterns. By co-designing the network architecture and the control systems, RDC can benefit a wide range of realistic data center workloads. Our evaluations with testbed and simulation setups show that RDC leads to substantial performance benefits for real-world applications.

Acknowledgment

We thank our shepherd George Porter and the anonymous reviewers for their valuable feedback. This research is partly sponsored by the NSF under CNS-1718980, CNS-1801884, and CNS-1815525.

References

- [1] Top of rack vs end of row data center designs. <http://bradhedlund.com/2009/04/05/top-of-rack-vs-end-of-row-data-center-designs/>, 2009.
- [2] S320 photonic switch hardware user manual. <http://www.calient.net/wp-content/uploads/downloads/2013/04/CALIENT-S-Series-Photonic-Switch-Hardware-User-Manual-Rev-A-460xxx-00-v10.pdf>, 2012.
- [3] Introducing data center fabric, the next-generation facebook data center network. <https://code.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network>, 2014.
- [4] Facebook network analytics data sharing. <https://www.facebook.com/groups/1144031739005495/>, 2016.
- [5] Apache hadoop: Fair scheduler. <https://hadoop.apache.org/docs/r2.7.4/hadoop-yarn/hadoop-yarn-site/FairScheduler.html>, 2017.
- [6] Sailing through the data deluge. <https://rockleyphotonics.com/wp-content/uploads/2019/02/Rockley-Photonics-Sailing-through-the-Data-Deluge.pdf.>, 2019.
- [7] 25.6 tb/s strataxgs broadcom tom-ahawk 4 ethernet switch series. <https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series>, 2020.
- [8] 320x320 3D MEMS optical circuit switch. <https://www.calient.net/products/edge640-optical-circuit-switch/>, 2020.
- [9] 32*100Gbps Ethernet Switch. <https://www.fs.com/products/107081.html>, 2020.
- [10] Apache hadoop. <http://hadoop.apache.org>, 2020.
- [11] Apache hadoop: Capacity scheduler. <https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html>, 2020.
- [12] Apache hadoop yarn project. <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>, 2020.
- [13] Hdfs architecture. <https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>, 2020.
- [14] Specifying data center it pod architectures. https://www.apc.com/salestools/WTOL-AHAPRN/WTOL-AHAPRN_R0_EN.pdf, 2020.
- [15] 100G PAM4 850nm 100m optical transceiver module. <https://www.fs.com/products/93264.html>, 2021.
- [16] 32*400Gbps Ethernet Switch. <https://www.fs.com/products/96982.html>, 2021.
- [17] 400G QSFP-DD Passive Direct Attach Copper Twinax Cable (3m). <https://www.fs.com/products/82454.html>, 2021.
- [18] Barefoot tofino. <https://www.barefootnetworks.com/products/brief-tofino>, 2021.
- [19] Core and pod data center design. <http://go.bigswitch.com/rs/974-WXR-561/images/Core-and-Pod%20Overview.pdf>, 2021.
- [20] Duplex single mode optical fiber cable (10m). <https://www.fs.com/products/40203.html>, 2021.
- [21] Duplex single mode optical fiber cable (3m). <https://www.fs.com/products/40193.html>, 2021.
- [22] Ibm prefabricated modular data center. <https://www.ibm.com/us-en/marketplace/prefabricated-modular-data-center>, 2021.
- [23] Memcached. <https://memcached.org>, 2021.
- [24] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.
- [25] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 1–13, 2014.
- [26] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.
- [27] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 89–92, 2010.

- [28] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.
- [29] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.
- [30] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation NSDI 12*, pages 253–266, 2012.
- [31] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.
- [32] Paraskevas Bakopoulos, Konstantinos Christodoulopoulos, Giada Landi, Muzzamil Aziz, Eitan Zahavi, Domenico Gallico, Richard Pitwon, Konstantinos Tokas, Ioannis Patronas, Marco Capitani, et al. Nephele: An end-to-end scalable and dynamically reconfigurable optical architecture for application-aware sdn cloud data centers. *IEEE Communications Magazine*, 56(2):178–188, 2018.
- [33] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, et al. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 782–797, 2020.
- [34] Hitesh Ballani, Paolo Costa, Istvan Haller, Krzysztof Jozwik, Kai Shi, Benn Thomsen, and Hugh Williams. Bridging the last mile for optical switching in data centers. In *Optical Fiber Communication Conference*, pages W1C–3. Optical Society of America, 2018.
- [35] Joshua L Benjamin, Thomas Gerard, Domanic Lavery, Polina Bayvel, and Georgios Zervas. Pulse: optical circuit switched data center architecture operating at nanosecond timescales. *Journal of Lightwave Technology*, 38(18):4906–4921, 2020.
- [36] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.
- [37] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 65–72. ACM, 2009.
- [38] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 8. ACM, 2011.
- [39] Sergey Blagodurov, Alexandra Fedorova, Evgeny Vinik, Tyler Dwyer, and Fabien Hermenier. Multi-objective job placement in clusters. In *SC’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.
- [40] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. *ACM SIGCOMM Computer Communication Review*, 42(4):431–442, 2012.
- [41] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. {TAO}: Facebook’s distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.
- [42] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A distributed and robust sdn control plane for transactional network updates. In *2015 IEEE conference on computer communications (INFOCOM)*, pages 190–198. IEEE, 2015.
- [43] Andromachi Chatzieleftheriou, Sergey Legtchenko, Hugh Williams, and Antony Rowstron. Larry: Practical network reconfigurability in the data center. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 141–156, 2018.
- [44] Kai Chen, Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, Xitao Wen, and Yan Chen. Osa: An optical switching architecture for data center networks with unprecedented flexibility. *IEEE/ACM Transactions on Networking*, 22(2):498–511, 2014.

- [45] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 231–242. ACM, 2013.
- [46] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM Computer Communication Review*, 41(4):98–109, 2011.
- [47] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 443–454, 2014.
- [48] Kari Clark, Hitesh Ballani, Polina Bayvel, Daniel Cletheroe, Thomas Gerard, Istvan Haller, Krzysztof Jozwik, Kai Shi, Benn Thomsen, Philip Watts, et al. Sub-nanosecond clock and data recovery in an optically-switched data centre network. In *2018 European Conference on Optical Communication (ECOC)*, pages 1–3. IEEE, 2018.
- [49] Kari A Clark, Daniel Cletheroe, Thomas Gerard, Istvan Haller, Krzysztof Jozwik, Kai Shi, Benn Thomsen, Hugh Williams, Georgios Zervas, Hitesh Ballani, et al. Synchronous subnanosecond clock and data recovery for optically switched data centres using clock phase caching. *Nature Electronics*, 3(7):426–433, 2020.
- [50] Sushovan Das, Weitao Wang, and TS Ng. Towards all-optical circuit-switched datacenter network cores: The case for mitigating traffic skewness at the edge. In *ACM SIGCOMM 2021 Workshop on Optical Systems (OptSys’ 21)*, 2021.
- [51] Mauro Dell’Amico and Silvano Martello. Bounds for the cardinality constrained p cmax problem. *Journal of Scheduling*, 4(3):123–138, 2001.
- [52] Vojislav Dukic, Ginni Khanna, Christos Gkantsidis, Thomas Karagiannis, Francesca Parmigiani, Ankit Singla, Mark Filer, Jeffrey L Cox, Anna Ptasznik, Nick Harland, et al. Beyond the mega-data center: networking multi-data center regions. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 765–781, 2020.
- [53] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. *ACM SIGCOMM Computer Communication Review*, 40(4):339–350, 2010.
- [54] G.C Fox, S.W Otto, and A.J.G Hey. Matrix algorithms on a hypercube i: Matrix multiplication. *Parallel Computing*, 4(1):17 – 31, 1987.
- [55] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.
- [56] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 249–264, 2016.
- [57] Thomas Gerard, Kari Clark, Adam Funnell, Kai Shi, Benn Thomsen, Philip Watts, Krzysztof Jozwik, Istvan Haller, Hugh Williams, Paolo Costa, et al. Fast and uniform optically-switched data centre networks enabled by amplitude caching. In *2021 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3. IEEE, 2021.
- [58] Thomas Gerard, Christopher Parsonson, Zacharaya Shabka, Polina Bayvel, Domanıç Lavery, and Georgios Zervas. Swift: Scalable ultra-wideband subnanosecond wavelength switching for data centre networks. *arXiv preprint arXiv:2003.05489*, 2020.
- [59] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 216–229. ACM, 2016.
- [60] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi*, volume 11, pages 24–24, 2011.
- [61] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.
- [62] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. VI2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

- [63] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.
- [64] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *ACM SIGCOMM Computer Communication Review*, 39(4):63–74, 2009.
- [65] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 75–86. ACM, 2008.
- [66] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyash Sekar, Samir R Das, Jon P Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 319–330. ACM, 2014.
- [67] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM ’17*, pages 29–42, New York, NY, USA, 2017. ACM.
- [68] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Measuring control plane latency in sdn-enabled switches. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 25. ACM, 2015.
- [69] Christian E Hopps. Analysis of an equal-cost multi-path algorithm. 2000.
- [70] Chang-Hong Hsu, Qingyuan Deng, Jason Mars, and Lingjia Tang. Smoothoperator: Reducing power fragmentation and improving power utilization in large-scale datacenters. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’18*, pages 535–548, New York, NY, USA, 2018. ACM.
- [71] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.
- [72] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. *ACM SIGCOMM Computer Communication Review*, 45(4):407–420, 2015.
- [73] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. *ACM SIGCOMM Computer Communication Review*, 44(4):539–550, 2014.
- [74] Srikanth Kandula, Jitendra Padhye, and Paramvir Bahl. Flyways to de-congest data center networks. 2009.
- [75] Robert Krauthgamer, Joseph Naor, and Roy Schwartz. Partitioning graphs into balanced components. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 942–949. SIAM, 2009.
- [76] Maciej Kuźniar, Peter Perešini, and Dejan Kostić. What you need to know about sdn flow tables. In *International Conference on Passive and Active Network Measurement*, pages 347–359. Springer, 2015.
- [77] Sophie Lange, Arslan S Raja, Kai Shi, Maxim Karпов, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Fotini Karinou, Xin Fu, Junqiu Liu, et al. Sub-nanosecond optical switching using chip-based soliton microcombs. In *Optical Fiber Communication Conference*, pages W2A–4. Optical Society of America, 2020.
- [78] Dominique LaSalle and George Karypis. Multi-threaded graph partitioning. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 225–236. IEEE, 2013.
- [79] T Li, B Cole, P Morton, and D Li. Rfc2281: Cisco hot standby router protocol (hsrp), 1998.
- [80] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with reactor. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 1–15, Seattle, WA, 2014. USENIX Association.
- [81] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zupdate: Updating data center networks with zero loss. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 411–422. ACM, 2013.
- [82] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Presented as part of the 10th*

- USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, 2013.
- [83] Vincent Liu, Danyang Zhuo, Simon Peter, Arvind Krishnamurthy, and Thomas Anderson. Subways: A case for redundant, inexpensive data center edge links. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 27. ACM, 2015.
- [84] Yunpeng James Liu, Peter Xiang Gao, Bernard Wong, and Srinivasan Keshav. Quartz: a new design element for low-latency dcns. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 283–294. ACM, 2014.
- [85] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. *arXiv e-prints*, page arXiv:1903.12307, Mar 2019.
- [86] William M Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 267–280. ACM, 2017.
- [87] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.
- [88] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28. ACM, 2017.
- [89] Wil Michiels, Jan Korst, Emile Aarts, and Jan Van Leeuwen. Performance ratios for the differencing method applied to the balanced number partitioning problem. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 583–595. Springer, 2003.
- [90] Tal Mizrahi and Yoram Moses. Time4: Time for sdn. *IEEE Transactions on Network and Service Management*, 13(3):433–446, 2016.
- [91] Samuel K Moore. Another step toward the end of moore’s law: Samsung and tsmc move to 5-nanometer manufacturing-[news]. *IEEE Spectrum*, 56(6):9–10, 2019.
- [92] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and sharing in disaggregated {Rack-Scale} storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 17–33, 2017.
- [93] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.
- [94] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–12, 2018.
- [95] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. *research.microsoft.com*, 2011.
- [96] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM ’13, pages 447–458, New York, NY, USA, 2013. ACM.
- [97] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. *ACM SIGCOMM Computer Communication Review*, 41(4):266–277, 2011.
- [98] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. *ACM SIGCOMM Computer Communication Review*, 42(4):323–334, 2012.
- [99] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network’s (datacenter) network. *SIGCOMM Comput. Commun. Rev.*, 45(4):123–137, August 2015.
- [100] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. In-band synchronization for distributed sdn control planes. *ACM SIGCOMM Computer Communication Review*, 46(1):37–43, 2016.
- [101] Tae Joon Seok, Niels Quack, Sangyo Han, Wencong Zhang, Richard S Muller, and Ming C Wu. Reliability study of digital silicon photonic mems switches. In *2015 IEEE 12th International Conference on Group IV Photonics (GFP)*, pages 205–206. IEEE, 2015.

- [102] Tae Joon Seok, Niels Quack, Sangyo Han, Wencong Zhang, Richard S Muller, and Ming C Wu. Reliability study of digital silicon photonic mems switches. In *Group IV Photonics (GFP), 2015 IEEE 12th International Conference on*, pages 205–206. IEEE, 2015.
- [103] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiyi Zhang. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.
- [104] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 255–270, Boston, MA, 2019. USENIX Association.
- [105] 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.
- [106] Ankit Singla, Chi-Yao Hong, Lucian Popa, and Philip Brighten Godfrey. Jellyfish: Networking data centers, randomly. In *NSDI*, volume 12, pages 1–6, 2012.
- [107] Rob Stone, Ruby Chen, Jeff Rahn, Srinivas Venkataraman, Xu Wang, Katharine Schmidtke, and James Stewart. Co-packaged optics for data center switching. In *2020 European Conference on Optical Communications (ECOC)*, pages 1–3. IEEE, 2020.
- [108] Xiongchao Tang, Haojie Wang, Xiaosong Ma, Nosayba El-Sayed, Jidong Zhai, Wenguang Chen, and Ashraf Aboulnaga. Spread-n-share: improving application performance and cluster throughput with resource-aware job placement. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2019.
- [109] Meg Walraed-Sullivan, Amin Vahdat, and Keith Marzullo. Aspen trees: balancing data center fault tolerance, scalability and cost. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 85–96, 2013.
- [110] Guohui Wang, David G Andersen, Michael Kaminsky, Konstantina Papagiannaki, TS Ng, Michael Kozuch, and Michael Ryan. c-through: Part-time optics in data centers. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 327–338. ACM, 2010.
- [111] Dingming Wu, Weitao Wang, Ang Chen, and TS Ng. Say no to rack boundaries: Towards a reconfigurable pod-centric dcn architecture. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 112–118. ACM, 2019.
- [112] Dingming Wu, Yiting Xia, Xiaoye Steven Sun, Xin Sunny Huang, Simbarashe Dzinamarira, and TS Eugene Ng. Masking failures from application performance in data center networks with shareable backup. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 176–190, 2018.
- [113] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 419–430. ACM, 2012.
- [114] Yiting Xia, Xin Sunny Huang, and T. S. Eugene Ng. Stop rerouting!: Enabling sharebackup for failure recovery in data center networks. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, HotNets-XVI*, pages 171–177, New York, NY, USA, 2017. ACM.
- [115] Yiting Xia, Xiaoye Steven Sun, Simbarashe Dzinamarira, Dingming Wu, Xin Sunny Huang, and TS Eugene Ng. A tale of two topologies: Exploring convertible data center network architectures with flat-tree. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 295–308, 2017.
- [116] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.
- [117] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 139–150. ACM, 2012.
- [118] Xia Zhou, Zengbin Zhang, Yibo Zhu, Yubo Li, Saipriya Kumar, Amin Vahdat, Ben Y Zhao, and Haitao Zheng. Mirror mirror on the ceiling: Flexible wireless links for data centers. *ACM SIGCOMM CCR*, 42(4):443–454, 2012.

- [119] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 362–375, 2017.
- [120] Christopher Zimmer, Saurabh Gupta, Scott Atchley, Sudharshan S Vazhkudai, and Carl Albing. A multi-faceted approach to job placement for improved performance on extreme-scale systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1015–1025. IEEE, 2016.

A Appendix

This appendix includes more discussions and results.

A.1 The RDC 0/1 updates

Instead of changing packet version, in RDC a switch performs binary changes from VLAN tagging packets to not, and from not VLAN tagging packets to tagging. Assume packets are in VLAN tagging mode before the change and there is a single VLAN tagging rule at the ingress switch for all packets. We first install the new set of rules with lower priority that matches only on destination IPs, note that the more general matching rules always have lower priority. Then, we remove the VLAN tagging rule. The untagged packets in the transient state can immediately match against the new set of rules. Similarly, if packets are not in VLAN tagging mode before the update, we first install the new set of rules matches on both VLAN tag and destination IPs and then install a single VLAN tagging rule for all packets.

Fig. 16 illustrates the update mechanism in RDC, which we call 0/1 update. It uses an example of forwarding state updates on an OpenFlow ToR switch, which has 4 ports. Ports 1 and 2 are connected to servers, ports 3 and 4 are connected to the agg. switches. Packet versions are encoded in the VLAN tag. Before the update, packets are first matched against a VLAN table that tags packets with a VLAN ID. Those tagged packets are then matched against the old rules in the forwarding table. During the transient state of rule updating, packets become untagged and can thus immediately match against the new rules without being dropped. The instructions of the forwarding table direct packets to the group table where packets are either directly sent out via an output port or get load-balanced over multiple output ports using the *select* group type. Similarly, an update from the not-tagging mode to the tagging mode also causes no packet loss.

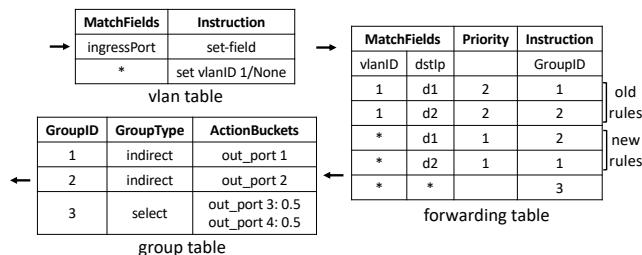


Figure 16: An example of RDC’s 0/1 rule update on an OpenFlow-enabled ToR switch.

A.2 Use case: Uplink load-balancing

In §4, we have discussed four use cases for RDC. Among these, uplink load balancing is another reactive RDC algorithm. We discuss this use case in more detail, including the data collection, bias mitigation, and control algorithm. The proactive mode (Use case 3) is simply driven by applications,

and the mixed optimization (Use case 4) is also case-specific, so we focus on the reactive algorithm for Use case 2.

Traffic data collection. RDC maintains flow counters on ToRs to monitor the amount of traffic that each server has sent outside the pod. We assume each RDC pod has a unique ID, e.g., an IP address prefix shared by all servers in the pod. Counters are only installed and updated for inter-pod traffic. This can be implemented in the switch using two separate flow tables. The first flow table matches on the destination IP prefix and has only one rule matching the switch’s own pod ID. If the first table misses, the second table then matches the 5-tuple and updates the associated counters. Otherwise, the packet skips the second table and goes to the forwarding table. By default, a miss on the second table will not result in packet loss, but a *go-to* action to the rest of the switch pipeline, which avoids traffic disruption when the counter rules change.

Demand estimation. We use a similar technique to estimate the true demand of servers in bottlenecked racks assuming they fair-share the uplink bandwidth. The estimates are obtained by first aggregating the flow counters for each server and then scaling up the per-server demand to reach an aggregate uplink throughput as if the rack is not oversubscribed. We only apply this technique to racks that have been bottlenecked in the collection period to prevent idle racks from being mistakenly treated as hot. This technique keeps the relative order of server traffic load but brings larger quantitative differences among servers, guiding our algorithm to compute better topologies.

Algorithm. For 1-CS RDC, we view the uplink load-balancing problem as a balanced graph partition problem; for multi-CS RDC, we use a heuristic algorithm to obtain the reconfiguration plan. The details of the above two algorithms are included in §A.4.

A.3 Hedera demand estimation algorithm

The pseudocode is shown in Algorithm 1. M is the demand matrix, H is the set of hosts in the network. e_S is the equal share rate of the flows, d_T is the total demand for the destination, and d_S is the demand limited by the sender, $f.rl$ is a flag for a receiver-limited flow, and $\langle src \rightarrow anydst \rangle$ represents all the flows from the specific source host src to any destination host.

A general explanation for this algorithm is expanding the flow demand at the source host with the fair share, and then reducing the demand of some flows according to the capacity of the destination hosts. In each iteration, one or more flows will converge. Eventually, all the flows will converge after multiple iterations [27].

A.4 Topology optimization algorithm details

1. Problem formulation. Assume that the number of racks in a pod is m , each rack has n servers, and each pod has k CS switches to reallocate the server. To keep a record of which server is connected to which ToR switch, we use another

Algorithm 1: Hedera demand estimation [27]

Input: M : traffic matrix, H : the set of all hosts
Output: M : estimated demand matrix

```

1 while some  $M_{i,j}$  demand changed do
2   for host  $src \in H$  do
3      $es \leftarrow \frac{1 - \sum \text{converged flow demand}}{\text{unconverged flow number}}$ ,
        flow  $\in < src \rightarrow \text{anydst} >$ 
4     for flow  $f \in < src \rightarrow \text{anydst} >$  do
5       if  $f$  not converged then
6          $M_{f.src,f.dst}.\text{demand} \leftarrow es$ 
7   for host  $dst \in H$  do
8     for  $f \in < \text{anysrc} \rightarrow dst >$  do
9        $f.rl \leftarrow \text{true}$ 
10       $d_T \leftarrow d_T + f.\text{demand}$ 
11       $n_R \leftarrow n_R + 1$ 
12    if  $d_T > 1$  then
13       $es \leftarrow \frac{1}{n_R}$ 
14      while some  $f.rl$  was set to false do
15         $n_R \leftarrow 0$ 
16        for  $f \in < \text{anysrc} \rightarrow dst > \& f.rl$  do
17          if  $f.\text{demand} < es$  then
18             $d_S \leftarrow d_S + f.\text{demand}$ 
19             $f.rl \leftarrow \text{false}$ 
20          else
21             $n_R \leftarrow n_R + 1$ 
22         $es \leftarrow \frac{1 - d_S}{n_R}$ 
23        for  $f \in < src \rightarrow dst > \& f.rl$  do
24           $M_{f.src,f.dst}.\text{demand} \leftarrow es$ 
25           $M_{f.src,f.dst}.\text{converged} \leftarrow \text{true}$ 
```

matrix $C[mn][m]$, if $C[i][j]$ is 1, then server i is connected to ToR j ; the server and TOR are not connected if the value is 0. For a valid allocation of the servers, the first constraint is that one server should only be connected to one ToR:

$$\sum_{j=0}^{m-1} C[i][j] = 1, \forall i \in [0, mn] \quad (1)$$

The second constraint is because only a limited number of ports from each ToR are connected to every CS, which is $\frac{n}{k}$. Thus, among all the $\frac{mn}{k}$ servers connected to one CS, only $\frac{n}{k}$ of them can be connected to the same ToR switch:

$$\sum_{x=0}^{m-1} \sum_{y=0}^{\frac{n}{k}-1} C[x \cdot n + i * \frac{n}{k} + y][j] = \frac{n}{k}, \forall i \in [0, k], \forall j \in [0, m] \quad (2)$$

The goal for traffic localization is to localize the inter-rack

traffic within a pod as much as possible. Hence, the objective function is to maximize the total amount of localized traffic. Assume that the traffic demand matrix is $D[mn][mn]$, which covers all the server pairs in a pod. Only when two servers are connected to the same ToR, $C[x][j] \cdot C[y][j] = 1$, so that the following equation shows the amount of localized traffic demand:

$$\text{Maximize: } \sum_{x=0}^{mn-1} \sum_{y=0}^{mn-1} \sum_{j=0}^{m-1} C[x][j] \cdot C[y][j] \cdot D[x][y] \quad (3)$$

The goal for uplink load-balancing is to balance the load across all uplinks. Hence, we choose to minimize the maximum load of any uplink for the out-of-pod traffic. Assume that the out-of-pod traffic demand matrix is $U[mn]$. The objective function is:

$$\text{Minimize: } \text{MAX} \left\{ \sum_{i=0}^{mn-1} C[i][j] \cdot U[i] \right\}, j \in [0, m] \quad (4)$$

2. Heuristic traffic localization algorithm for 1-CS RDC. For RDC with only 1 circuit switch, the topology optimization problem will just become a graph partition problem. And the new objective function is that we want to partition the vertices (servers) in the graph into groups equally and let the edges (traffic demand) within the groups to be maximum. Assume the traffic demand is a graph $G = (E, V)$, where V is the vertex set (i.e., servers) and E is the edge set. The weight of an edge e , $w(e)$, is the traffic demand between the vertices. To simplify the computation, we do not distinguish the directions of traffic between a server pair, i.e., graph G is non-directional. Our goal is to partition the graph into subgraphs of equal numbers of vertices such that the weighted sum of cross-subgraph edges is minimized. We require partitions of the same size because each ToR must connect to the same fixed number of servers. The balanced graph partitioning problem is NP-hard, but high-quality, efficient heuristics have been proposed in a library *parmetis* [78]. Thus, for the traffic localization problem, we can set the objective to maximize the edge weights insides each group and use the BGP method to solve it.

3. Heuristic uplink load-balancing algorithm for 1-CS RDC. For RDC with only 1 circuit switch, the uplink load-balancing problem will also become a graph partition problem. Our formulation partitions mn number of servers $1, 2, \dots, mn$ into m subsets S_1, S_2, \dots, S_m such that each subset S_j has exactly n servers and the maximum cost of a subset, defined as $\text{max}(\{c(S_j)\})$ is minimized, where $c(S_j) = \sum U[i](i \in S_j)$. Again, we require a balanced partition of the servers because each ToR must host the same number of servers. The problem is also NP-hard when $k > 2$ [51, 89]. We use the same high-quality and efficient heuristics, *parmetis*, to solve this problem by simply changing the objective function to balance the out-of-pod throughput for each group.

4. Heuristic traffic localization algorithm for multi-CS

RDC. For RDC with multiple circuit switches, our heuristic firstly groups the servers under the same CS into m bundles equally and maximizes the traffic within each bundle, since all the servers within a bundle should be connected to the same ToR. After obtaining the bundles, for one CS, we only need to assign each of them to a different ToR switch, and the goal is to maximize the traffic demand among bundles under the same ToR switch. In total we have mk bundles, each bundle will be connected to one ToR switch, recorded as $BC[mk][m]$. Moreover, the bundles can be used to calculate an aggregated traffic demand matrix $BD[mk][mk]$. Thus, the simplified traffic localization algorithm can be presented as:

$$\sum_{j=0}^{m-1} BC[i][j] = 1, \forall i \in [0, mk) \quad (5)$$

$$\sum_{x=0}^{m-1} BC[x \cdot m + i][j] = 1, \forall i \in [0, m), \forall j \in [0, m) \quad (6)$$

$$\text{Maximize: } \sum_{x=0}^{mk-1} \sum_{y=0}^{mk-1} \sum_{j=0}^{m-1} BC[x][j] \cdot BC[y][j] \cdot BD[x][y] \quad (7)$$

5. Heuristic uplink load-balancing algorithm for multi-CS RDC.

For the heuristic ULB algorithm, again the servers are grouped under the same CS into m bundles equally. And the objective function is to minimize the maximum out-of-pod traffic of each bundle. The idea behind this heuristic is that if each OCS gives balanced out-of-pod traffic to each ToR, then the total out-of-pod traffic from each ToR should also be balanced. The constraints remain the same. Thus, we divide the problem into many sub-problems, and each sub-problem focuses on the servers connected to the same OCS:

$$\sum_{j=0}^{m-1} C[i][j] = 1, \forall i \in [0, mn) \quad (8)$$

$$\sum_{x=0}^{m-1} \sum_{y=0}^{\frac{n}{k}-1} C[x \cdot n + i * \frac{n}{k} + y][j] = \frac{n}{k}, \forall i \in [0, k), \forall j \in [0, m) \quad (9)$$

$$\text{Minimize: } MAX \left\{ \sum_{r=0}^{m-1} \sum_{i=0}^{\frac{n}{k}-1} C[r * n + \frac{n}{k} * s + i][j] \cdot U[i] \right\}, \forall j \in [0, m) \quad (10)$$

Isolation Mechanisms for High-Speed Packet-Processing Pipelines

Tao Wang[†]

Xiangrui Yang^{‡*}

Gianni Antichi^{**}

Anirudh Sivaraman[†]

Aurojit Panda[†]

[†]*New York University* [‡]*National University of Defense Technology*

^{**}*Queen Mary University of London*

Abstract

Data-plane programmability is now mainstream. As we find more use cases, deployments need to be able to run multiple packet-processing modules in a single device. These are likely to be developed by independent teams, either within the same organization or from multiple organizations. Therefore, we need isolation mechanisms to ensure that modules on the same device do not interfere with each other.

This paper presents Menshen, an extension of the Reconfigurable Match Tables (RMT) pipeline that enforces isolation between different packet-processing modules. Menshen is comprised of a set of lightweight hardware primitives and an extension to the open source P4-16 reference compiler that act in conjunction to meet this goal. We have prototyped Menshen on two FPGA platforms (NetFPGA and Corundum). We show that our design provides isolation, and allows new modules to be loaded without impacting the ones already running. Finally, we demonstrate the feasibility of implementing Menshen on ASICs by using the FreePDK45nm technology library and the Synopsys DC synthesis software, showing that our design meets timing at a 1 GHz clock frequency and needs approximately 6% additional chip area. We have open sourced the code for Menshen’s hardware and software at <https://isolation.quest/>.

1 Introduction

Programmable network devices in the form of programmable switches [6, 15, 26] and smart network interface cards (SmartNICs) [10, 11, 44] are becoming commodity. Such devices allow the network infrastructure to provide its users additional services beyond packet forwarding, e.g., congestion control [41, 66], measurement [52], load balancing [62], in-network caches [60], and machine learning [72].

As network programmability matures, a single device will have to concurrently support *multiple* independently developed modules. This is the case for *networks in the public cloud* where tenants can provide packet-processing

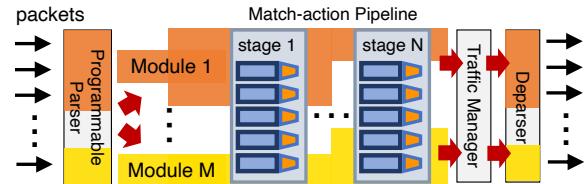


Figure 1: The RMT architecture [36] typically consists of a programmable parser/deparser, match-action pipeline and traffic manager. Menshen provides isolation between RMT modules. In the figure we show resources allocated to *module 1* and *module m* by shading them in the appropriate color.

modules that are installed and run on the cloud provider’s devices. Another example is when different teams in an organization write different modules, e.g., an in-networking caching module and a telemetry module.

Isolation is required to safely run multiple modules on a single device. Several prior projects have observed this need and proposed solutions targeting multicore network processors [50, 68], FPGA-based packet processors [63, 73, 77, 82], and software switches [53, 81]. However, thus far, high-speed pipelines such as RMT that are used in switch and NIC ASICs provide only limited support for isolation. For instance, the Tofino programmable switch ASIC [26] provides mechanisms to share stateful memory across modules but cannot share other resources, e.g., match-action tables [79].

Our goal with this paper is to lay out the requirements for isolation mechanisms on the RMT architecture that are applicable to all resources and then to design lightweight mechanisms that meet these requirements. As presented in Figure 1, the desired isolation mechanisms should guarantee that multiple modules can be allocated to different resources, and process packets in parallel without impacting each other. In brief (§2.1 elaborates), we seek isolation mechanisms that ensure that (1) one module’s behavior (input, output, and internal state) is unaffected by another module; (2) one module can not affect another’s throughput and/or latency; and (3) one module can not access RMT pipeline resources belonging to another. Given the high performance

*Work done at Queen Mary University of London

requirements of RMT, we also seek mechanisms that are lightweight. Finally, the isolation mechanism should ensure that one module can be updated without disturbing any other modules and that the update process itself is quick.

The RMT architecture poses unique challenges for isolation because its pipeline design means that neither an OS nor a hypervisor can be used to enforce isolation.¹ This is because RMT is a *dataflow* or spatial hardware architecture [34, 39] with a set of instructions units continuously processing data (packets). This is in contrast to the Von Neumann architecture found on processors [27], where a program counter decides what instruction to execute next. As such, an RMT pipeline is closer in its hardware architecture to an FPGA or a CGRA [70] than a processor. This difference in architecture has important implications for isolation. The Von Neumann architecture supports a *time-sharing* approach to isolation (in the form of an OS/hypervisor) that runs different modules on the CPU successively by changing the program counter to point to the next instruction of the next module. We instead use *space-partitioning* to divide up the RMT pipeline’s resources (e.g., match-action tables) across different modules.

Unfortunately, space partitioning is not a viable option for certain RMT resources because there are very few of them to be effectively partitioned across modules (e.g., match key extraction units (§3.1)). For such resources, we add additional hardware primitives in the form of small tables that store module-specific configurations for these resources. As a packet progresses through the pipeline, the packet’s module identifier is used as an index into these tables to extract module-specific configurations before processing the packet according to the just extracted configuration. These primitives are similar to the use of *overlays* [3, 16] in embedded systems [1, 25] and earlier PCs [17]. They effectively allow us to bring in different configurations for the same RMT resource, in response to different packets from different modules.

Based on the ideas of space partitioning and overlays, we build a system, Menshen, for isolation on RMT pipelines. Specifically, Menshen makes the following contributions:

1. The use of space partitioning and overlays as techniques to achieve isolation when sharing an RMT pipeline across multiple modules.
2. A hardware design for an RMT pipeline that employs these techniques.
3. An implementation on 2 open-source FPGA platforms: the NetFPGA switch [84] and Corundum NIC [45].
4. A compiler based on the open-source P4-16 compiler [18] that supports multiple modules running on RMT, along with a system-level module to provide basic services (e.g., routing, multicast) to other modules.
5. An evaluation of Menshen using 8 modules—based on tutorial P4 programs, and the NetCache [60] and NetChain [59] research projects—showing that

¹An OS does run on the network device’s control CPU, allowing isolation in the control plane. Our focus, instead, is on isolation in the data plane.

Menshen meets our isolation requirements.

6. An ASIC analysis of the Menshen, which shows that our design can meet timing at 1 GHz (comparable to current programmable ASICs) with modest additional area relative to a baseline RMT design.

Overall, we find that Menshen adds modest overhead to an existing RMT pipeline in both FPGA and ASIC implementations (§5). Our main takeaway is that a small number of simple additions to RMT along with changes to the RMT compiler can provide inter-module isolation for a high-speed packet-processing pipeline. We have made Menshen’s hardware design and software available under an open-source license at <https://isolation.quest/> to enable further research into isolation mechanisms for high-speed pipelines.

2 The case for isolation

A single network device might host a measurement module [52], a forwarding module [74], an in-network caching [60] module, and an in-network machine-learning module [72]—each written by a different team in the same organization. It is important to isolate these modules from each other. This would prevent bugs in measurement, in-network caching, and in-network ML from causing network downtime. It would also ensure that memory for measuring per-flow stats [65] is separated from memory for routing tables, e.g., a sudden arrival of many new flows does not cause cached routes to be evicted from the data plane.

The packet-processing modules in question do not even have to be developed by teams in the same organization [79]. They could belong to different tenants sharing the same public cloud network. This would allow cloud providers to offer network data-plane programmability as a service to their tenants, similar to cloud CPU, GPU, and storage offerings today. Such a capability would allow tenants to customize network devices in the cloud to suit their needs.

2.1 Requirements for isolation mechanisms

For the rest of this paper, we will use the term module to refer to a packet-processing program that must be isolated from other such programs, regardless of whether the modules belong to different mutually distrustful tenants or to a single network operator. Importantly, modules can not call each other like functions, but are intended to isolate different pieces of functionality from each other—similar to processes. Based on our use cases above (§2), we want an inter-module isolation mechanisms that meet the requirements below:

1. **Behavior isolation.** The behavior of one module must not affect the behavior (i.e., input, output, computation and internal state) of another. This would prevent a faulty or malicious module from adversely affecting other modules. Further, one module should not be able to inspect the behavior of another module.
2. **Resource isolation.** A switch/NIC pipeline has multiple resources, e.g., static random-access memory (SRAM)

for exact matching and ternary content-addressable memory (TCAM) for ternary matching. Each module should be able to access only its assigned subset of the pipeline’s resources and no more. It should also be possible to allocate each resource independent of other resources. For example, an in-network caching module may need large amounts of stateful memory [60] for its caches, but a routing module may need significant TCAM for routing tables.

3. **Performance isolation.** Each module should stay within its allotted ingress packets per second and bits per second rates. One module’s behavior should not affect the throughput and latency of another module.
4. **Lightweight.** The isolation mechanisms themselves must have low overhead so that their presence does not significantly degrade the high performance of the underlying network device. In addition, the extra hardware consumed by these mechanisms must be small.
5. **Rapid reconfiguration.** If a module is reconfigured with new packet-processing logic, the reconfiguration process should be quick.
6. **No disruption.** If a module is reconfigured, it must not disrupt the behavior of other unchanged modules—especially important in a multi-tenant environment [40].

2.2 Target setting for Menshen

We target both programmable switches and NICs with a programmable packet-processing pipeline based on the RMT pipeline [36], a common architecture for packet processing for the highest end devices. Other projects have looked at isolation for software switches, multicore network processors, FPGA-based devices, and the Barefoot Tofino switch (without hardware changes). §6 compares against them.

An RMT pipeline can be implemented either on an FPGA (e.g., FlowBlaze [71], Lightning NIC [57], nanoPU [56]) or an ASIC (e.g., the Tofino [26], Spectrum [15], and Trident [6] switches; and the Pensando NIC [13]). This pipeline has also been embedded within larger hardware designs (e.g., PANIC [67]). Menshen builds on a baseline RMT pipeline to provide isolation between different modules/tenants. A high-speed implementation of Menshen would likely be based on an RMT ASIC. For this paper, we prototype RMT on 2 FPGA-based platforms: the NetFPGA switch [84] and the Corundum NIC [45]. Our ASIC synthesis results suggest that our lessons generalize to ASICs as well (§5.2).

3 Design

In order to meet its performance goals, RMT’s pipelined architecture ensures that processing stages never stall, i.e., they can process a packet every clock cycle. The Menshen design aims to preserve this invariant so that isolation does not come at the cost of performance. To maintain this invariant, Menshen’s isolation mechanisms cannot reconfigure stages or change table contents between packets. As a result,

Applied Mechanism	Targeted Resources
Space partitioning	Match action table entries, stateful memories
Overlays	Parsing actions, key extractors, packet header vector (PHV) containers, arithmetic logic units (ALUs)

Table 1: Summary of Menshen’s mechanisms.

Menshen provides isolation by spatially partitioning switch resources between packet processing modules.

While spatial partitioning is easy for resources, e.g., match-action tables and stateful memory, that are provisioned so they can be allocated at flow granularity, it is much more challenging for resources such as key extractors (§3.1) which are generally shared across flows. This is because naive approaches to spatially partitioning such shared resources across packet-processing modules would severely reduce the number of resources available to each packet processing module—and hence the richness of that module.

To see why, consider a case where a key extractor is split between two packet processing modules: in this setting each packet processing module can only use half the key extractor, limiting its key length to half of what it would be able to use were it running on the entire pipeline. This problem is of course further exacerbated as we increase the number of packet processing modules sharing the pipeline.

Menshen addresses this problem using *overlays*: we associate a configuration lookup table with each shared resource in the switch. This lookup table is keyed by the packet processing module’s ID and contains the configuration that should be used when processing packets for this module. For example, in the case of the key extractor, the configuration table contains the instructions that the module uses to construct key (§3.1). Our use of overlays means that we do not need to partition resources including ALUs or PHVs between modules. Instead, the module has exclusive access to all PHVs/ALUs in a stage when processing a packet. Table 1 summarizes our mechanisms.

To realize Menshen, on the software side, we modify an RMT compiler to target a block of resources rather than the entire pipeline. Overlays require new hardware primitives to be added to the RMT pipeline. These hardware primitives are small tables that contain per-module configurations of shared resources. On every packet, these tables are indexed using the packet’s module ID to determine the configuration to use for that packet at that resource. An incremental deployment pathway for Menshen would be to only modify an RMT compiler (e.g., Tofino’s compiler) to implement space partitioning without investing in new overlay hardware.

3.1 Menshen hardware

The Menshen hardware design (Figure 2) builds on RMT by adding hardware primitives for isolation into the RMT pipeline. Because these isolation primitives are added

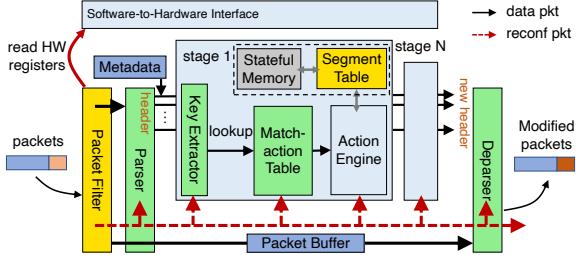


Figure 2: Menshen hardware and software-to-hardware interface. Menshen builds on a RMT [36] pipeline, by adding Yellow components and modifying Green ones.

pervasively throughout the pipeline, we first describe the overall Menshen hardware design including both RMT and the new isolation primitives. We then summarize the new isolation primitives added by Menshen.

Menshen expects that a data packet’s header carries information identifying what module should process the packet. Currently in our prototype, this is the VLAN ID (VID) header, which we assume is set by the vSwitch [51], but other fields, e.g., VxLAN ID, can be used instead. Packets entering Menshen are first handled by a packet filter that discards packets without a VLAN ID.² Next, a parser extracts the VLAN ID from the packet and applies module-specific parsing to extract module-specific headers from the TCP/UDP payload. The parser then pushes these parsed packet headers into packet header vector (PHV) containers that travel through the pipeline of match-action stages.

Each stage forms keys out of headers, looks up the keys in a match-action table, and performs actions. At the start of each stage, a key extractor in the stage forms a key by combining together the headers in a module-specific manner. The keys are then concatenated with the module ID and looked up in a match-action table, whose space is partitioned across different modules. If the key matches against a match-action pair in the table, the lookup result is used to index an action table.

Similar to the match-action table, the action table is also partitioned across modules. Each action in the table identifies opcodes, operands, and immediate constants for a very-large instruction word (VLIW), controlling many parallel arithmetic and logic units (ALUs). The VLIW instruction consumes the current PHV to produce a new PHV as input for the next stage. The table’s action can modify persistent pipeline state, stored in stateful memory. Stateful memory is indexed by a physical address that is computed from a local address, obtained from a module’s packets. This computation is done by a segment table, which stores the offset and range of each module’s slice of stateful memory. We now detail the main components of our design.

Parser. The Menshen parser is driven by a table lookup process similar to the RMT parser [36, 49]. Specifically, whenever a new packet comes in, the module ID is extracted

²The filter can be configured to send control packets without VLAN tags, e.g., BFD packets [5], to the control plane or system-level module (§3.3).

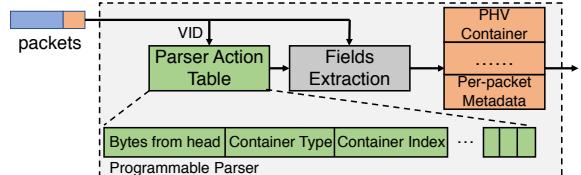


Figure 3: Menshen programmable parser.

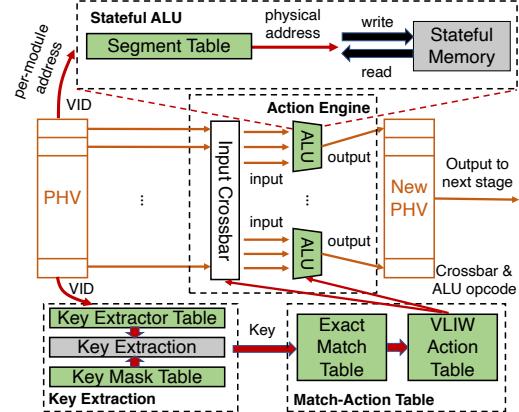


Figure 4: Menshen processing stage.

from its VLAN ID prior to parsing the rest of the packet. This module ID is then used as an index into the table that determines how to parse the rest of the packet (Figure 3). Each table entry corresponds to multiple parsing actions for a module—one action per extracted PHV container. Each parsing action specifies (1) *bytes from head*, indicating where in the packet the parser should extract a particular header, (2) *container type* (e.g., 4-byte container, etc.), indicating how many bytes we should extract; (3) *container index*, indicating where in the PHV we should put the extracted header into. The parser also sets aside space in the pipeline (e.g., time of enqueue into switch output queues and queueing delay after dequeue) and for temporary packet headers used for computation.

Key extractor. Before a stage performs a lookup on a match-action table, a lookup key must be constructed by extracting and combining together one or more PHV containers. This key extraction process differs between modules in the same stage, and between different stages for the same module. To implement key extraction, just like the parser, we use a key extractor table (Figure 4) that is indexed by a packet’s module ID. Each entry in this table specifies which PHV containers to combine together to form the key. These PHV containers are then selected into the key using a multiplexer for each portion of the key. To enable variable-length key matching for different modules, the key extractor also includes a key mask table, which also uses the module ID as an index to determine how many bits to pad in the key to bring it up to a certain fixed key size before lookup.

Match table. Each stage looks up the fixed-size key con-

structed by the key extractor in a match table. Currently, we support only exact-match lookup. The match table is statically partitioned across modules by giving a certain number of entries to each module. To enforce isolation among different modules, the module ID is appended to the key output by the key extractor. This augmented key is what is actually looked up against the entries in the match table; each entry stores both a key and the module ID that the key belongs to. The lookup result is used as index into the VLIW action table to identify a corresponding action to execute.

Action table and action engine. Each VLIW action table entry indicates which fields from the PHV to use as ALU operands (i.e., the configuration of each ALU’s operand crossbar) and what opcode should be used for each ALU controlled by the VLIW instruction (i.e., addition, subtraction, etc.). Each ALU outputs a value based on its operands and opcode. There is one ALU per PHV container, removing the need for a crossbar on the output because each ALU’s output is directly connected to its corresponding PHV container. After a stage’s ALUs have modified its PHV, the modified PHV is passed to the next stage.

Stateful memory. Menshen’s action engines can also modify persistent pipeline state on every packet. Each module is assigned its own address space, and the available stateful memory in Menshen is partitioned across modules. When a module accesses its slice of stateful memory, it supplies a per-module address that is translated into a physical address by a segment table before accessing the stateful memory. To perform this translation, Menshen stores per-module configuration (i.e., base address and range) in a segment table, which can be indexed by the packet’s module ID. Menshen borrows this idea of a segment table from NetVRM’s [79, 83] page table, but implements it in hardware instead of programming it in P4 atop Tofino’s stateful memory like NetVRM does. This allows Menshen to avoid using scarce Tofino stateful memory to emulate a segment table. Also, by adding segment table hardware to each stage, Menshen avoids sacrificing the first stage of stateful memory for a segment table, instead reclaiming it for useful packet processing. This is unlike NetVRM, which can share stateful memory across modules only from the second stage because the first stage is used for the page table.

Deparser. The deparser performs the inverse operation of the parser. It takes PHV containers and writes them back into the appropriate byte offset in the packet header, merges the packet header with the corresponding payload in the packet buffer, and transmits the merged packet out of the pipeline. The format of the deparser table is identical to the parser table and is similarly indexed by a module ID.

Secure reconfiguration. Our threat model assumes that the Menshen hardware and software are trusted, but that data packets that enter the Menshen pipeline are untrusted. Data packets are untrusted because for a switch, they can come from physical machines outside the switch’s control and,

for a NIC, they can come from tenant VMs sharing the NIC. Hence, the pipeline should be reconfigured only by Menshen software, not data packets.

This is a security concern faced by existing RMT pipelines as well, even without isolation support. Commercial programmable switches solve this problem by using a separate daisy chain [7] to configure pipeline stages. This chain carries configuration commands that are picked up by the intended pipeline stage as the command passes that stage. The chain is only accessible over PCIe, which is connected to the control-plane CPU, but not by Ethernet ports, which carry outside data packets. Hence, the only way to *write* new configurations into the pipeline is through PCIe. The packet-processing pipeline is restricted to just *reading* configurations and using them to implement packet processing. Thus, the daisy chain provides secure reconfiguration by physically separating reconfiguration and packet processing.

Menshen uses a similar approach by employing a daisy chain for reconfiguration when a module is updated. A special *reconfiguration packet* carries configuration commands for the pipeline’s resources (e.g., parser). Our implementation of this daisy chain varies depending on the platform. For our NetFPGA prototype, this daisy chain is connected solely to the switch CPU via PCIe, similar to current switches. For our Corundum NIC prototype, we connect the daisy chain directly to PCIe and use a *packet filter* before our parser to filter out reconfiguration packets from untrusted data packets by ensuring that reconfiguration packets have a specific UDP destination port. An ideal solution would be to use a physically separate interface, e.g., USB or JTAG, for reconfiguring the Menshen pipeline on Corundum, but we found it challenging to implement such a physically separate reconfiguration interface on Corundum. In Appendix A, we show how a daisy chain permits more rapid reconfiguration than an alternative approach of using the AXI-L protocol on an FPGA.

Summary of Menshen’s new primitives. The hardware primitives introduced by Menshen on top of an RMT pipeline (Figure 2) are the configuration tables for the parser, deparser, key extractor, key mask units and segment table. These tables provide an overlay feature to share the same unit across multiple modules. Specifically, for each unit, Menshen provides a table with a configuration entry per module, rather than one configuration for the whole unit. In addition, Menshen introduces the packet filter to ensure secure reconfiguration. Menshen also modifies match tables, by appending the module ID to the match key and the match-action entries. Finally, Menshen partitions match-action tables and stateful memory across all modules. These primitives ensure that updating one module only affects a single entry (for Menshen resources that use overlays) and only affects a subset of memory (for Menshen resources that use space partitioning), thus allowing us to update one module without disrupting others (§2.1).

ASIC feasibility of Menshen’s primitives. Menshen’s parser, deparser, key extractor, key mask, and segment tables are

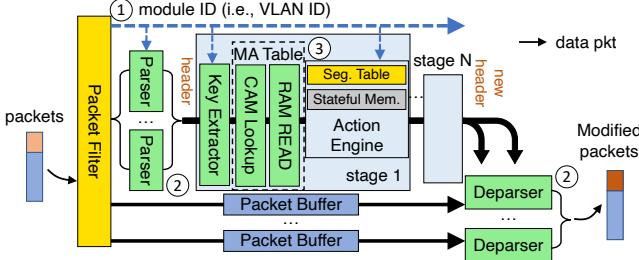


Figure 5: Three optimization techniques applied in Menshen. Numbered circles refer to specific techniques in §3.2.

small and simple arrays indexed by the module identifier. They can be readily realized in SRAM that can support a memory read every clock cycle. The packet filter is a simple combinational circuit that checks if the incoming packet is destined to a specific UDP destination port. Extending the match-action tables in each stage to append a module ID to every entry amounts to modestly increasing the key width in the table. While these new primitives add some additional latency relative to RMT, e.g., to go through the packet filter or reading out the per-module parser configuration, the pipelined nature of RMT means that this additional latency does not impact packet-forwarding rate. The ASIC area overhead increases as we increase the number of simultaneous programming modules that need to be supported; we quantify it in §5.2.

3.2 Improving Menshen’s throughput

As shown in Figure 5, we apply 3 main techniques to optimize the forwarding performance of Menshen: (1) masking RAM read latency, (2) using multiple parsers and deparsers, and (3) increasing pipeline depth. We demonstrate the effect these techniques have on Menshen’s throughput in §5.2.

① Masking RAM read latency. The design described in §3.1 attaches the module ID to the PHV that is sent from one element (e.g., parser, key extractor) to the next. In this design, we read the module’s configuration from SRAM after the PHV arrives, thus incurring a few additional clock cycles of latency. To optimize this, we mask SRAM access latency by splitting the module ID from the PHV and sending the module ID to the next element *ahead of time*. The PHV follows the module ID, and thus the module configuration at a stage can be read concurrently with the PHV being transmitted to that stage.

② Multiple parsers and deparsers. In §3.1’s design, there is one parser, deparser, and packet buffer. The parser extracts and parses the header and puts the full packet in the packet buffer. Then the deparser takes the modified headers from the last stage, uses them to overwrite the relevant portions of the full packet in the packet buffer, and sends out the packet.

Our optimized design uses multiple parallel parsers, deparsers, and packet buffers to improve throughput. Deparsing is the most expensive operation as any position within the PHV container might be modified, and thus any part of the packet header (128 bytes in our implementation) might need

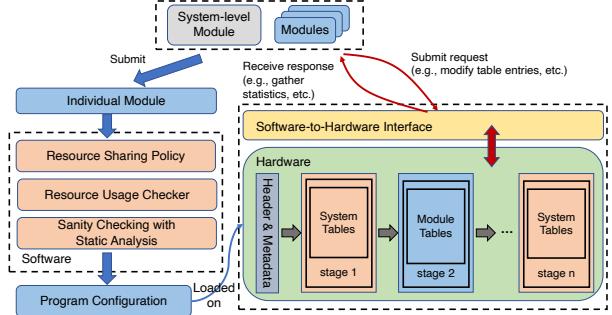


Figure 6: Menshen software and system-level module.

to be updated. Furthermore, deparsing has to process both the packet header and the payload. Therefore, we use 4 parallel deparsers and 2 parsers. We also associate a separate packet buffer with each deparser.

On ingress, the packet filter tags each packet with a packet buffer number (0–3) in round robin order. It also round robins incoming packets to the 2 parsers. The last pipeline stage uses the packet buffer tag to determine which packet buffer’s packet the last stage’s modified PHV should be combined with. Each packet buffer’s deparser combines the earliest packet from the packet buffer along with the last stage’s most recently modified PHV for that buffer.

③ Deep pipelining. With careful digital design, in Menshen’s implementation, we can pipeline each element (e.g., match-action table) into several sub elements to improve throughput. For example in Figure 5, we divide the match-action table into CAM-lookup and action-RAM-read sub elements. In this specific example, this allows us to process a PHV every 2 clock cycles at each sub-element rather than every 4 clock cycles at the whole match-action table.

3.3 The Menshen system-level module

To hide information about the underlying physical infrastructure (e.g., topology) from tenant modules in a virtualized environment, modules in Menshen can use virtual IP addresses to operate in a shared environment [51]. Here, virtual IP addresses are local and scoped to modules belonging to a particular tenant, regardless of which physical device these modules are on. To support virtual IPs and provide basic services to other modules, Menshen contains a system-level module written in P4-16 that provides common OS-like functionality, e.g., converting virtual IPs to physical IPs, multicast, and looking up physical IPs to find output ports. The system-level module has 3 benefits: (1) it avoids duplication among different modules re-implementing common functions, improving the resource efficiency of the pipeline, (2) it hides underlying physical details (e.g., topology) from each module so that one tenant’s modules on different network devices can form a virtual network [51], and (3) it provides common and useful real-time statistics (e.g., link utilization, queue length, etc.) that can inform packet processing within modules.

Figure 6 shows how the system-level module is laid out

relative to the other modules. Packets entering the Menshen pipeline are first processed by the system-level module before being handed off to their respective module for module-specific processing. After module-specific processing, these packets enter the system module for a second time before exiting the pipeline. The first time they enter the system-level module, packets can read and update system-level state (e.g., link utilization, packet counters, queue measurements), whereas the second time they enter the system-level module, module-specific packet header fields (e.g., virtual IP address) can be read by the system-level module to determine device-specific information (e.g., output port). In both halves, there is a narrow interface by which modules communicate with the system-level module. This split structure of the system-level module arises directly from the feed-forward nature of the RMT pipeline, where packets typically only flow forward, but not backward. Hence, packets pick up information from the system-level module in the first stage and pass information to the system-level module in the last stage. The non-system modules are sandwiched in between these two halves.

3.4 Menshen software

The software-hardware interface. The Menshen software-to-hardware interface works similar to P4Runtime [19] to support interactions (e.g., modifying match-action entries, fetching hardware statistics, etc.) between the Menshen software and the Menshen hardware. However, in addition to P4Runtime’s functions, Menshen’s software-hardware interface can also be used to reconfigure different hardware resources (Appendix C) in Menshen to reprogram them when a module is added or updated. This allows us to dynamically reconfigure portions of Menshen as module logic changes.

The Menshen resource checker. The Menshen resource checker ensures that each module’s resource allocation complies with an operator specified resource sharing policy (e.g., dominant resource sharing (DRF) [48], or a utility-based [54] policy). In our current design we check allocations statically because reassigning resources from one module to another disrupts processing for both modules. Instead we rely on admission control and do not load a module whose resource requirements cannot be met. We leave the question of what is an appropriate resource allocation policy to future work.

The Menshen static checker. To ensure isolation, Menshen’s static checker analyzes 3 properties of the module’s P4 source code. First, it checks that modules do not modify hardware-related statistics (e.g., link utilization) provided by the system-level module to all modules. Second, modules can not modify their VID. This is because a module can be spread across multiple programmable devices [46, 59], and changes to VIDs by module A on a device can unintentionally affect a module B on a downstream device, where B’s real VID happens to be the same as A’s modified VID. Third, modules must not recirculate packets and their routing tables

should be loop-free.³ This is because all modules share the same ingress pipeline bandwidth. Recirculating packets or looping them back through multiple devices will degrade the ability of other modules to process packets.

The Menshen compiler. Packet-processing pipelines (e.g., RMT [36]) are structured as feed-forward pipelines of programmable units, each of which has limited processing capabilities. This design ensures the *all-or-nothing* property: once a module has been compiled and loaded it can run at up to line rate, while modules that can not run at line rate cannot be compiled. Menshen’s compiler follows the same design, and only admits modules that meet line-rate requirements.

The compiler reuses the frontend and midend of the open-source P4-16 reference compiler [18] and creates a new backend similar to BMv2 [4]. This backend has a parser, a single processing pipeline, and a deparser. The compiler takes a module’s P4-16 program as input and conducts all the resource usage and static checks described above. Then, for the parser and deparser, it transforms the parser defined in the module to configuration entries for the parser and deparser tables. For the packet-processing pipeline, which consists of match-action tables, it transforms the key in a table to a configuration in the key extractor table, and actions to VLIW action table entries according to the opcodes. The compiler also performs dependency checking [36, 61] to guarantee that all ALU actions and key matches are placed in the proper stage, respecting table dependencies.

The Menshen compiler can be extended to support the same packet flowing through different P4 modules belonging to one tenant. The compiler can take multiple P4 modules as input, assign them the same module ID, and allocate them to non-overlapping pipeline stages—similar to how we lay out user and system modules in different stages as in Figure 6.

3.5 Limitations

As a research prototype, Menshen has several limitations. First, while we have developed *mechanisms* to support isolation across multiple modules, we have not yet designed *policies* that decide how much of each resource a module should be given [35]. Second, our FPGA implementation of RMT lacks many features present in a commercial RMT implementation such as the Barefoot Tofino switch [26]. Third, our compiler currently does not perform any compiler optimizations for code generation [47] or memory allocation [46, 61]. Fourth, Menshen proposes isolation mechanisms for the packet-processing pipeline, but does not deal with isolating traffic from different modules competing for output link bandwidth, which is a orthogonal traffic management problem. Proposals like PIFO [75] can be used here, by assigning PIFO ranks to different modules to realize a desired inter-module bandwidth-sharing policy.

³We check loop freedom in the control plane.

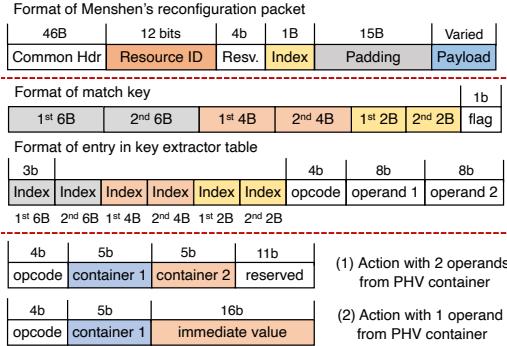


Figure 7: Formats of Menshen’s packets and tables.

4 Implementation

4.1 Menshen hardware

To implement Menshen, we first built a baseline RMT implementation for an FPGA. Menshen includes (1) a packet filter to filter out reconfiguration packets from data packets using a specific predefined UDP destination port (i.e., 0x1f2), (2) a programmable parser, (3) a programmable RMT pipeline with 5 programmable processing stages, (4) a deparser, and (5) a separate daisy-chain pipeline for reconfiguration. It also includes Menshen’s primitives for isolation. We have integrated it into both the Corundum NIC [45] and the NetFPGA reference switch [84]. The Menshen code base together with the optimizations (§3.2) consists of 9975 lines of Verilog. Of this, 3098 and 3226 lines are for handling data bus widths of 512 bits (Corundum) and 256 bits (NetFPGA) respectively. 3651 lines are for the common blocks, e.g., key extractor, etc. Below, we describe our hardware implementation in more detail. Figure 7 shows the formats of Menshen’s packets and tables.

PHV format. Menshen’s PHV has 3 types of containers of different sizes, namely 2-byte, 4-byte and 6-byte containers. Each type has 8 containers. Also, we allocate and append an additional 32 bytes to store platform-specific metadata (e.g., an indication to drop the packet, destination port, etc.), which results in a PHV length of 128 bytes in total. Thus, we have a total of $3 \times 8 + 1 = 25$ PHV containers. To prevent any possibility of PHV contents leaking from one module to another, the PHV is zeroed out for each incoming packet.

Reconfiguration packet format. Figure 7 shows the format of Menshen reconfiguration packets. The reconfiguration packet is a UDP packet with the standard UDP, Ethernet, VLAN, and IP headers. Within the UDP payload, a 12-bit resource ID indicates which hardware resource within which stage should be updated (e.g., key extractor table in stage 3). To reconfigure the resource, the table storing the configuration for this resource must be updated by writing the entry stored within the reconfiguration packet’s payload at the location specified by the 1-byte index field in the reconfiguration packet header. The UDP destination port field determines whether the reconfiguration packet is valid or not.

Operation	Description
add/sub	Add/subtract between containers
addi/subi	Add/subtract an immediate to/from container
set	Set a container to an immediate value
load	Load a value from stateful memory
store	Store a value to stateful memory
loadd	Load value from stateful memory, add 1, and store back
port	Set destination port
discard	Discard packet

Table 2: Supported operations in Menshen’s ALU.

Packet filter. The packet filter has 2 registers that can be accessed by the Menshen software via Xilinx’s AXI-Lite protocol [28]: (1) a 4-byte reconfiguration packet counter, which monitors how many reconfiguration packets have passed through the daisy chain; (2) a 32-bit bitmap, which indicates which module is currently being updated (e.g., bit 1 stands for module 1, bit 2 for module 2, etc.). During reconfiguration of a module, via the software-to-hardware interface, the Menshen software reads the reconfiguration packet counter. It then writes the bitmap to reflect the module ID M of the module currently being updated. The bitmap is then consulted on every packet to drop data packets from M until reconfiguration completes, so that M ’s “in-flight” packets aren’t incorrectly processed by partial configurations.

Then, the Menshen software sends all reconfiguration packets embedded with the predefined UDP destination port to the daisy chain. Finally, it polls the reconfiguration packet counter to check if reconfiguration is over and then zeroes the bitmap so that M ’s packets are no longer dropped. Reconfiguration packets maybe dropped before they reach the RMT pipeline. This can be detected by polling the reconfiguration packet counter to see if it has been correctly incremented or not. If it hasn’t been incremented correctly, then the entire reconfiguration process restarts with M ’s packets being dropped until reconfiguration is successful.

Programmable parser/deparser. We currently support per-module packet header parsing in the first 128 bytes of the packet. These 128 bytes also include the headers common to all modules (e.g., Ethernet, VLAN, IP, and UDP). We design the parser action for each parsed PHV container as a 16-bit action. The first 3 bits are reserved. The next 7 bits indicate the starting extraction position in bytes from byte 0. These 7 bits can cover the whole 128-byte length. Then, the next 2 bits and 3 bits indicate the container type (2, 4, or 6 byte) and number (0–7) respectively. The last bit is the validity bit. For each module, we allocate 10 such parser actions (i.e., to parse out at most 10 containers), resulting in a 160-bit-wide entry for the parser action table.

We note that we only parse out fields of a packet into PHV containers, if those fields are actually used as part of either keys or actions in match-action tables. Before packets are sent out, the deparser pulls out the full packet (including the payload) from the packet buffer and only updates the portions of the packet that were actually modified by table actions. This approach allows us to reduce the number of PHV containers to

25 because packet fields that are never modified or looked up by the Menshen pipeline need not travel along with the PHV.

Key extractor. The key for lookup in the match-action table is formed by concatenating together up to 2 PHV containers each of the 2-byte, 4-byte, and 6-byte container types. Hence the key can be up to 24 bytes and 6 containers long. Since there are 8 containers per type, the key extraction table entry for each module in each stage uses $\log_2(8) * 6 = 18$ bits to determine which container to use for the 6 key locations. Additionally, the key extractor is also used to support conditional execution of actions based on the truth value of a predicate of the form $A \text{ } OP \text{ } B$, where A and B are packet fields and OP is a comparison operator. For this purpose, each key extractor table entry also specifies the 2 operands for the comparison operation and the comparison opcode. The opcode is a 4-bit number, while the operands are 8 bits each. The operands can either be an immediate value or refer to one of the PHV containers. The result of the predicate evaluation adds one bit to the original 24 byte key, bringing the total key length to $24 * 8 + 1 = 193$ bits. Because not all keys need to be 193 bits long, we use a 193-bit-wide mask table. Each entry in this table denotes the validity of each of the 193 key bits for each module in each stage. This is somewhat wasteful and can be improved by storing validity information within the key extractor table itself.

Exact match table. To implement the exact match table, we leverage the Xilinx CAM block [31]. This CAM matches the key from the key extractor module against the entries within the CAM. As discussed in §3.1, to ensure isolation between different modules, we append the module ID (i.e., VLAN ID) to each entry, which means that the CAM has a width of $193 + 12 = 205$ bits. The lookup result from the CAM is used to index the VLIW action table. The action is designed in a 25-bit format per ALU/container (Figure 7). As we have $24 + 1 = 25$ PHV containers, the width of the VLIW action table is $25 * 25 = 625$ bits. The Xilinx CAM block simplifies implementation of an exact-match table and can also easily support ternary matches if needed (Appendix B).

Action engine. The crossbar and ALUs in the action engine use the VLIW actions to generate inputs for each ALU and carry out per-ALU operations. ALUs support simple arithmetic, stateful memory operations (e.g., loads and stores), and platform-specific operations (e.g., discard packets) (Table 2). The formats of these actions are shown in Figure 7. Additionally, in stateful ALU processing, each entry in the segment table is a 2-byte number, where the first byte and second byte indicate memory offset and range, respectively.

Menshen primitives. Menshen’s isolation primitives (e.g., key-extractor and segment tables) are simple arrays implemented using the Xilinx Block RAM [30] feature.

4.2 Menshen Software

The Menshen compiler reuses the open-source P4-16 reference compiler [18] and implements a new backend

Program	Description
CALC [20]	return value based on parsed opcode and operands
Firewall [20]	stateless firewall that blocks certain traffic
Load Balancing [20]	steer traffic based on 4-tuple header info
QoS [20]	set QoS based on traffic type
Source Routing [20]	route packets based on parsed header info
NetCache [60]	in-network key-value store
NetChain [59]	in-network sequencer
Multicast [20]	multicast based on destination IP address

Table 3: Evaluated use cases.

extension in 3773 lines of C++. It takes the module written in P4-16 together with resource allocation as the inputs, and generates per-module configurations for Menshen hardware. Specifically, it (1) conducts resource usage checking to ensure every program’s resource usage is below its allocated amount; (2) places the system-level module’s (120 lines of P4-16) configurations in the first and last stages in the Menshen pipeline; and (3) allocates PHV containers to the fields shared between the system-level and other modules so that the other modules can be sandwiched between the two halves of the system-level module (§3.4). The Menshen software-to-hardware interface is written in Python. It configures Menshen hardware by converting program configurations to reconfiguration packets.

4.3 Corundum and NetFPGA integrations

We have integrated Menshen into 2 FPGA platforms: one for the NetFPGA platform that captures the hardware architecture of a switch [84], and another for the Corundum platform that captures the hardware architecture of a NIC [45]. Menshen’s integration on Corundum [45] is based on a 512-bit AXI-S [29] data width and runs at 250 MHz. Although Menshen’s pipeline can be integrated into both the sending and receiving path, in our current implementation, we have integrated Menshen into only Corundum’s sending path, i.e., PCIe input to Ethernet output. Menshen on NetFPGA [84] uses a 256-bit AXI-S [29] data width and runs at 156.25 MHz.

On the Corundum NIC platform, we insert a 1-bit discard flag, while on the NetFPGA switch platform, we insert a 1-bit discard flag and 128-bit platform-specific metadata, i.e., source port, destination port and packet length, into the PHV’s metadata field. A 4-bit one-hot encoded tag indicates the packet buffer (§3.2). The table depth in Menshen’s parser, key extractor, key mask, page, and deparser tables affects the maximum number of modules we can support and is currently 32. The depth of CAM and VLIW action table directly influences the amount of match-action entries and VLIW actions that can be allocated to all modules. Due to the open technical challenge of implementing CAMs on FPGAs efficiently [58, 71], we set their depth to 16 in each stage. While 16 is a small depth, the depth can be improved by using a hash table, rather than a CAM, for exact matching, e.g., cuckoo hashing [69].

5 Evaluation

In §5.1, we show that Menshen can meet our requirements (§2.1): it can be rapidly reconfigured, is lightweight, provides behavior isolation, and is disruption-free. Menshen achieves

performance isolation by (1) assuming packets exceed a minimum size (to guarantee line rate) and (2) forbidding recirculation. If either is violated, hardware rate limiters can be used to limit each module’s packet/bit rate. It achieves resource isolation by ensuring that a table entry for a resource (e.g., parser) is allotted to at most one module. In §5.2, we evaluate the current performance of Menshen.

Experimental setup. To demonstrate Menshen’s ability to provide multi-module support, we picked 6 tutorial P4 programs [20], as detailed in Table 3, together with simplified versions of NetCache [60] and NetChain [59].⁴ The system-level module provides basic forwarding and routing, with multicast logic integrated in it. Menshen’s parameters are detailed in §4 and summarized in Table 5 in the Appendix.

Testbed. We evaluate Menshen based on our Corundum and NetFPGA integrations as described in §4. For the switch platform experiments on NetFPGA, we use a single quad-port NetFPGA SUME board [14], where two ports are connected to a machine equipped with an Intel Xeon E5645 CPU clocked at 2.40 GHz and a dual-port Intel XXV710 10/25GbE NIC. For the NIC platform experiments on Corundum, we use a single Xilinx Alveo U250 board [2], where one port is with Menshen for the transmitting path and this port is connected to a 100 GbE NIC as the receiving path. Both setups are used to check Menshen’s correctness (§5.1). For NetFPGA performance tests (§5.2), we use the host as a packet generator. For Corundum performance tests (§5.2), we internally connect its receiving and transmitting path, and use the Spirent tester [22] to generate traffic. We depict our testing setup in Appendix D.

5.1 Does Menshen meet its requirements?

Menshen can be rapidly reconfigured. Reconfiguration time includes both the software’s compilation time (Figure 8) and the hardware’s configuration time (Figure 9); we evaluate each separately. When a module is compiled, the compiler needs to generate both configuration bits for various hardware resources as well as match-action entries for the tables the module looks up. These match-action entries can and will be overwritten by the control plane, but we need to start out with a new set of match-action entries for a module to ensure no information leaks from a previous module.

Hence, every time a module is compiled, the compiler also generates match-action entries. Within an exact match table, these entries must be different from each other to prevent multiple lookup results. As a result, Menshen’s compilation time increases with the number of match-action entries in the module (Figure 8). To contextualize this, Menshen’s compile times (few seconds) compare favorably to compile times for Tofino (~10 seconds for our use cases) and FPGA synthesis times (10s of minutes). We note that this is an imperfect comparison: our compiler performs fewer optimizations than

⁴Our versions of NetChain and NetCache do not include some features such as tagging hot keys.

Hardware Implementation	Slice LUTs	Block RAMs
NetFPGA reference switch	42325 (9.77%)	245.5 (16.7%)
RMT on NetFPGA	200573 (46.3%)	641 (43.6%)
Menshen on NetFPGA	200733 (46.34%)	641 (43.6%)
Corundum	61463 (3.56%)	349 (12.98%)
RMT on Corundum	235686 (13.63%)	316 (11.75%)
Menshen on Corundum	235903 (13.65%)	316 (11.75%)

Table 4: Resources used by 5-stage Menshen pipeline, on NetFPGA SUME and AU250 boards, compared with reference switch, Corundum NIC, and RMT.

either the Tofino or FPGA compilers and our targets are simpler. That said, compilation can happen offline, and hence it is not as time-sensitive compared to run-time reconfiguration.

To measure time taken for Menshen’s configuration post compilation, we vary the number of entries the Menshen software has to write into the pipeline.⁵ Also, as a comparison, we evaluate the cost of the Tofino run-time APIs from Tofino SDE 9.0.0 to insert match-action table entries for the CALC program. From Figure 9, we observe that the time spent in configuration of the hardware via Menshen’s software-to-hardware interface is similar to Tofino’s run-time APIs.

Menshen can reconfigure without disruption. To show Menshen can support disruption-free reconfiguration, we launch three CALC programs with fixed input packet rate, i.e., 5:3:2 ratio on a single link for module 1, 2 and 3, respectively. We use netmap-based tcprelay to generate total traffic of 9.3 Gbit/s on a 10 Gbit/s link. 0.5 seconds in, we start to reconfigure the first module to see if the packet processing of other modules has stalled or not. In Figure 10 we show the throughput achieved by each of three modules when reconfiguring module 1. We can observe that model 2 and 3 see no impact on their throughput. This demonstrates that Menshen provides performance isolation, and that it is feasible for a tenant to reconfigure their module without impacting other tenants. By contrast, updating a module on Tofino (§6) requires resetting the entire switch pipeline. Even with Tofino’s Fast Refresh [9], this leads to a 50 ms disruption of all servers (and their VMs) whose traffic is routed through the switch. This disruption can be significant in public cloud environments, and in many cases renders dynamic reconfiguration infeasible.

Menshen is lightweight. We list Menshen’s resource usage of logic and memory (i.e., LUTs and Block RAMs), including absolute numbers and fractions, in Table 4. For comparison, we also list the resource usage of the NetFPGA reference switch and the Corundum NIC. We believe that the additional hardware footprint of Menshen is acceptable for the programmability and isolation mechanisms it provides relative to the base platforms. The reason that Menshen uses more LUTs than Block RAMs is that Menshen leverages the Shift Register Lookup (SRL)-based implementation of Xilinx’s CAM IP [31]. We also compared with an RMT design, where we modified Menshen’s hardware to support only one module. Relative to RMT, Menshen incurs an extra

⁵Since the Menshen hardware can’t currently support so many entries (§4.3), we overwrite previously written entries to measure configuration time.

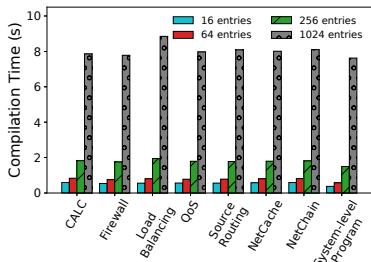


Figure 8: Compilation time.

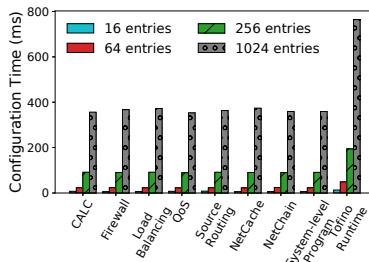


Figure 9: Configuration time.

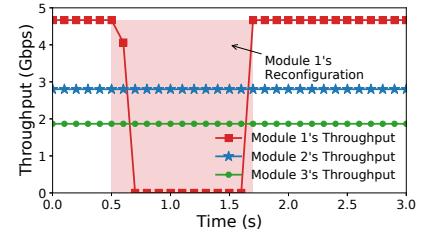


Figure 10: Throughput during reconfiguration.

0.65% (NetFPGA) and 0.15% (Corundum) in LUTs usage.

Menshen provides behavior isolation. Next, we spot check that Menshen can correctly isolate modules, i.e., every running module can concurrently execute its desired functionality. For this, we ran the CALC, Firewall, and NetCache module simultaneously on the Menshen pipeline. We generate data packets of different VIDs, which indicate which of these 3 modules they belong to, and input them to the Menshen FPGA prototype on both platforms. By examining the output packets at the end of Menshen’s pipeline, we checked that Menshen had correctly isolated the modules, i.e., each module behaved as it would have had it run by itself. We repeated the same experiment by running the Load Balancing, Source Routing, and NetChain modules simultaneously; we observed correct behavior isolation here too.

5.2 Menshen Performance

How many modules can be packed? In our current prototype on both Corundum and NetFPGA, we can support at most 32 modules because each isolation primitive (e.g., key extractor table) currently has 32 entries. In practice, the number of modules could be less than 32 if modules need to share a more bottlenecked hardware resource. For instance, if each module wants a match-action entry in every pipeline stage, the maximum number of modules is at most 16 because there are only 16 match-action entries in each stage in our current prototype. However, the numbers above are entirely a function of how much hardware one is willing to pay in exchange for multitenancy support. If we can afford to expend additional resources on an FPGA or extra area on an ASIC, we can correspondingly support a larger number of modules.

Latency. In our current implementation, the number of clock cycles needed to process a packet in the pipeline depends on packet size. This is because the number of cycles to process both the header and the payload depend on the header and payload length. For instance, for a minimum packet size of 64 bytes, Menshen’s pipeline introduces 79 and 106 cycles of processing for NetFPGA and Corundum, resulting in $79 * \frac{1000}{156.25} = 505.6$ ns and $106 * \frac{1000}{250} = 424$ ns latency, respectively. For the max. packet size of 1500 bytes, Menshen incurs 146 and 112 cycles for NetFPGA and Corundum, resulting in $150 * \frac{1000}{156.25} = 960$ ns and $129 * \frac{1000}{250} = 516$ ns latency.

Throughput. For NetFPGA, we used MoonGen [42] to generate packets with different sizes. Figure 11a shows that Menshen achieves a rate of 10 Gbit/s after a packet size of 96 bytes. This is the maximum supported by our MoonGen setup because we have a single 10G NIC. For Corundum, we internally connected Corundum’s receiving and transmitting path. Rather than using a host-based packet generator through PCIe, we used Spirent FX3-100GO-T2 tester to test Menshen’s throughput. The MTU size is set to 1500 bytes. As shown in Figure 11b and Figure 11c, optimized Menshen on Corundum achieves 100 Gbit/s at 256 bytes, while unoptimized Menshen can only achieve 80 Gbit/s at MTU-size packets. Also, we sample packets to evaluate the packet latency of optimized Menshen on Corundum with full rate. As depicted in Figure 11d, at full rate, it incurs about 1.2 μ s latency.

ASIC feasibility. With the same parameter settings in §5, we use the Synopsys DC synthesis tool [24] and FreePDK45nm technology library [8] to assess the ASIC feasibility of the Menshen pipeline.⁶ At 1 GHz frequency, when compared with an RMT design, where we modified Menshen to support only one module, Menshen incurs 18.5%, 7%, 20.9% additional chip area for the parser, deparser and one stage, respectively. For a 5-stage pipeline along with the packet filter, parser, deparser and packet buffers, Menshen (10.81 mm^2) incurs 11.4% additional chip area compared with RMT (9.71 mm^2).

Considering that memory (i.e., lookup tables) and packet processing logic only costs at most 50% in switch chip area [21, page 36], Menshen’s chip area overhead is moderate ($11.4\% * 50\% = 5.7\%$), which is conservative since the number of entries in our match-action table is only 16 (§4.1). With much larger number of entries in lookup tables—which is the common block between Menshen and RMT—Menshen’s additional chip area will be negligible.

6 Related work

Multi-core architecture solutions. To support isolation on programmable network devices based on multi-cores [10, 11, 23], FairNIC [50] partitions cores, caches, and memory across tenants and shares bandwidth across tenants through Deficit Weighted Round Robin (DWRR) scheduling.

⁶Since we can not have access to source code of Xilinx IPs (e.g., DMA, Ether+PHY, etc.), we solely run synthesis on Menshen’s Verilog codebase.

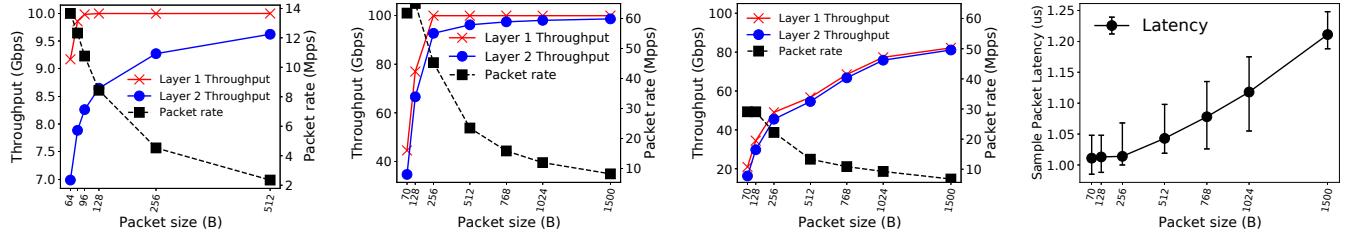


Figure 11: Results for performance benchmarks.

iPipe [68] uses a hybrid DRR+FCFS scheduler to share SmartNIC and host processors between different programs. Menshen uses space partitioning as well to allocate different resources to different modules. However, RMT’s spatial-/dataflow architecture differs considerably from the Von Neumann architectures for multi-core network processors targeted by FairNIC and iPipe. An RMT architecture can not support a runtime system similar to the ones used by iPipe and FairNIC.

FPGA-based solutions. Several FPGA platforms exist for programmable packet processing. These platforms can be broadly categorized into (1) direct programming of FPGAs [12, 44, 55, 64, 73, 77, 78] and (2) higher-level abstractions built on top of FPGAs [33, 37, 43, 71].

Systems (e.g., VirtP4 [73], MTPSA [77]) based on direct FPGA programming typically implement packet-processing logic in a hardware-description language (HDL) or using a high-level language like P4 [55, 78] or C [32, 64] that is translated into HDL. The HDL program is fed to an FPGA synthesis tool to produce a bitstream, which is written into the FPGA. This approach requires combining the programs of different modules into a single Verilog program, which can then be fed to the synthesis tool. Thus, changing one module disrupts other modules, violating our requirement of no disruption.

FlowBlaze [71], SwitchBlade [33], and hXDP [37] expose a restricted higher-level abstraction like RMT or eBPF on top of an FPGA. FlowBlaze and hXDP do not provide support for isolation. SwitchBlade does, but its higher-level abstraction is much less flexible than the RMT abstraction in Menshen. NICA [43] targets an FPGA NIC and is designed to share one pre-programmed offloading engine across many modules, while Menshen also targets ASIC pipelines and supports reprogramming individual modules without disrupting others.

Tofino [26]. Tofino is a commercial switch ASIC that uses multiple parallel RMT pipelines. However, Tofino currently does not support multiple modules/P4 programs within a single pipeline. The current Tofino compiler requires a single P4 program per pipeline. Multiple P4 programs can be merged into a single program per pipeline and then fed into the Tofino compiler (Wang et al. [79] and μ P4 [76]). However, both approaches still disrupt all tenants every time a single tenant in any pipeline is updated. This is because despite supporting an independent program per pipeline, updating any of these programs requires a reset of the entire Tofino switch [9].

Emulation-based solutions. Hyper4 [53] and HyperV [81] propose to emulate multiple P4 programs/modules using a single hypervisor P4 program, which can be configured at run time by the control plane, thus supporting disruption-free reconfiguration. However, we found that it was very challenging to design a sufficiently “universal” hypervisor program on a commercial RMT switch like Tofino.

As one example, the hypervisor program needs to support performing a bit-shift by an amount determined by a packet field, where the packet field is specified by the control plane. However, a high-speed chip like Tofino has several restrictions on bit-shifts and other computations for performance, e.g., on Tofino, the shift width and field to shift must be supplied at compile time, not at run time by the control plane.

PANIC [67] and FlexCore [80]. PANIC and FlexCore [80] are programmable multi-tenant NIC and switch designs, respectively. They both suffer from scalability issues because they need to build a large crossbar with long wires interconnecting all engines to each other, which requires careful physical design [38, Appendix C]. Menshen’s RMT pipeline is easier to scale as its wires are shorter: they only connect adjacent pipeline stages [36, 2.1].

7 Conclusion

This paper described Menshen, a system for isolating co-resident packet-processing modules on pipelines similar to RMT. Menshen builds on the idea of space partitioning and overlays, and is comprised of a set of simple hardware primitives that are inserted at different points in an RMT pipeline. These primitives are straightforward to realize in both ASICs and FPGAs. Menshen thus demonstrates that providing inter-module isolation in high-speed packet-processing pipelines is practical. Our software and hardware are available at <https://isolation.quest/>.

Acknowledgements. We thank the NSDI reviewers and our shepherd Rodrigo Fonseca for their insightful comments and suggestions. We also thank Mike Walfish, Ravi Netravali, Mina Tahmasbi Arashloo, Amy Oosterhout, and Fabian Ruffy for their suggestions on this paper. We thank Han Wang and Anurag Agrawal with whom we discussed the Tofino architecture, and Alex Forencich, the FlowBlaze and NetFPGA teams, who helped us with debugging and design. This work was funded in part by NSF grants CCF-2028832, CNS-2008048, UK EPSRC project EP/T007206/1, and a gift from Google.

References

- [1] About Arm Debugger Support for Overlays. <https://developer.arm.com/documentation/101470/2021-0/Debugging-Embedded-Systems/About-Arm-Debugger-support-for-overlays?lang=en>.
- [2] Alveo U250 Data Center Accelerator Card. <https://www.xilinx.com/products/boards-and-kits/alveo/u250.html>.
- [3] Before Memory was Virtual. <http://160592857366.free.fr/joe/ebooks/ShareData/Before%20Memory%20was%20Virtual%20By%20Peter%20J.%20Dunning%20from%20George%20Mason%20University.pdf>.
- [4] Behavioral model targets. https://github.com/p4lang/behavioral-model/blob/master/targets/R_EADME.md.
- [5] Bidirectional Forwarding Detection (BFD). <https://tools.ietf.org/html/rfc5880>.
- [6] BROADCOM Trident Programmable Switch. <https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series>.
- [7] Daisy Chain. [https://en.wikipedia.org/wiki/Daisy_chain_\(electrical_engineering\)](https://en.wikipedia.org/wiki/Daisy_chain_(electrical_engineering)).
- [8] FreePDK45. <https://www.eda.ncsu.edu/wiki/FreePDK45:Contents>.
- [9] Leveraging Stratum and Tofino Fast Refresh for Software Upgrades. https://opennetworking.org/wp-content/uploads/2018/12/Tofino_Fast_Refresh.pdf.
- [10] LiquidIO Smart NICs. <https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics.html>.
- [11] Mellanox BlueField VPI 100Gps SmartNIC. <https://www.mellanox.com/files/doc-2020/pb-bluefield-vpi-smart-nic.pdf>.
- [12] Mellanox Innova Open Programmable SmartNIC. <https://www.mellanox.com/sites/default/files/doc-2020/pb-innova-2-flex.pdf>.
- [13] Naples DSC-100 Distributed Services Card. https://pensando.io/assets/documents/Naples_100_ProductBrief-10-2019.pdf.
- [14] NetFPGA-SUME Virtex-7 FPGA Development Board. <https://reference.digilentinc.com/reference/programmable-logic/netfpga-sume/start>.
- [15] NVIDIA Mellanox Spectrum Switch. <https://www.mellanox.com/files/doc-2020/pb-spectrum-switch.pdf>.
- [16] Operating Systems Three Easy Pieces. <https://iitd-plos.github.io/os/2020/ref/os-arpaci-dessau-book.pdf>.
- [17] Overlaying in Commodore. https://www.atarimagazines.com/compute/issue73/loading_and_linking.php.
- [18] P4-16 Reference Compiler. <https://github.com/p4lang/p4c>.
- [19] P4 Runtime. <https://p4.org/p4-runtime/>.
- [20] P4 Tutorial. <https://github.com/p4lang/tutorials>.
- [21] Programmable Forwarding Planes are Here to Stay. <https://conferences.sigcomm.org/sigcomm/2017/files/program-netpl/01-mckeown.pptx>.
- [22] Spirent Quint-Speed High-Speed Ethernet Test Modules. https://assets.ctfassets.net/wcx9ap8i19s/12bhgz12JBkRa66QUG4N0L/af328986e22b1694b95b290c93ef6c21/Spirent_fX3_HSE_Module_datasheet.pdf.
- [23] Stingray SmartNIC Adapters and IC. <https://www.broadcom.com/products/ethernet-connectivity/smartnic>.
- [24] Synopsys DC Ultra. <https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html>.
- [25] The Space Shuttle Fight Software Development Process. <https://www.nap.edu/read/2222/chapter/5>.
- [26] Tofino: P4-programmable Ethernet switch ASIC. <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html>.
- [27] Von Neumann Architecture. https://en.wikipedia.org/wiki/Von_Neumann_architecture.
- [28] Xilinx AXI4-Lite Interface Protocol. <https://www.xilinx.com/products/intellectual-property/axi4-lite.html>.
- [29] Xilinx AXI4-Stream. https://www.xilinx.com/products/intellectual-property/axi4-stream_interconnect.html.

- [30] Xilinx Block Memory Generator v8.4. https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/pg058-blk-mem-gen.pdf.
- [31] Xilinx Parameterizable Content-Addressable Memory. https://www.xilinx.com/support/documentation/application_notes/xapp1151_Param_CAM.pdf.
- [32] Xilinx Vitis High-Level Synthesis. <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>.
- [33] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster. SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware. In *ACM SIGCOMM*, 2010.
- [34] K. Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE TC*, 1990.
- [35] M. Blöcher, L. Wang, P. Eugster, and M. Schmidt. Switches for HIRE: Resource Scheduling for Data Center in-Network Computing. In *ACM ASPLOS*, 2021.
- [36] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.
- [37] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *USENIX OSDI*, 2020.
- [38] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Varagaitik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. dRMT: Disaggregated Programmable Switching. In *ACM SIGCOMM*, 2017. Tech report available at https://cs.nyu.edu/~anirudh/sigcomm17_drm_t_extended.pdf.
- [39] J. B. Dennis and D. P. Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. In *ACM ISCA*, 1974.
- [40] S. Dharanipragada, S. Joyner, M. Burke, J. Nelson, I. Zhang, and D. R. K. Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems, 2021.
- [41] N. Dukkipati. *Rate Control Protocol (RCP): Congestion Control to Make Flows Complete Quickly*. PhD thesis, Stanford University, 2008.
- [42] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *ACM IMC*, 2015.
- [43] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *USENIX ATC*, 2019.
- [44] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI*, 2018.
- [45] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An Open-Source 100-Gbps NIC. In *IEEE FCCM*, 2020.
- [46] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*, 2020.
- [47] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta. Switch Code Generation Using Program Synthesis. In *ACM SIGCOMM*, 2020.
- [48] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*, 2011.
- [49] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design Principles for Packet Parsers. In *ACM/IEEE ANCS*, 2013.
- [50] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *ACM SIGCOMM*, 2020.
- [51] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM*, 2009.
- [52] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *ACM SIGCOMM*, 2018.

- [53] D. Hancock and J. van der Merwe. HyPer4: Using P4 to Virtualize the Programmable Data Plane. In *ACM CoNEXT*, 2016.
- [54] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker. Modular Switch Programming Under Resource Constraints. In *USENIX NSDI*, 2022.
- [55] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *ACM/SIGDA FPGA*, 2019.
- [56] S. Ibanez, A. Mallory, S. Arslan, T. Jepsen, M. Shahbaz, N. McKeown, and C. Kim. The nanoPU: Redesigning the CPU-Network Interface to Minimize RPC Tail Latency, 2020.
- [57] S. Ibanez, M. Shahbaz, and N. McKeown. The Case for a Network Fast Path to the CPU. In *ACM HotNets*, 2019.
- [58] W. Jiang. Scalable Ternary Content Addressable Memory Implementation Using FPGAs. In *ACM/IEEE ANCS*, 2013.
- [59] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX NSDI*, 2018.
- [60] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP*, 2017.
- [61] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling Packet Programs to Reconfigurable Switches. In *USENIX NSDI*, 2015.
- [62] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *ACM SOSR*, 2016.
- [63] J. Krude, J. Hofmann, M. Eichholz, K. Wehrle, A. Koch, and M. Mezini. Online Reprogrammable Multi Tenant Switches. In *1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, 2019.
- [64] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *ACM SIGCOMM*, 2016.
- [65] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*, 2016.
- [66] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu. HPCC: High Precision Congestion Control. In *ACM SIGCOMM*, 2019.
- [67] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *USENIX OSDI*, 2020.
- [68] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading Distributed Applications onto SmartNICs Using IPipe. In *ACM SIGCOMM*, 2019.
- [69] R. Pagh and F. F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 2004.
- [70] Y. Park, H. Park, and S. Mahlke. CGRA Express: Accelerating Execution Using Dynamic Operation Fusion. In *ACM CASES*, 2009.
- [71] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano. FlowBlaze: Stateful Packet Processing in Hardware. In *USENIX NSDI*, 2019.
- [72] A. Sapiro, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtarik. Scaling Distributed Machine Learning with In-Network Aggregation. In *USENIX NSDI*, 2021.
- [73] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja. Hard Virtualization of P4-Based Switches with VirtP4. In *ACM SIGCOMM Posters and Demos*, 2019.
- [74] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu. DC.P4: Programming the Forwarding Plane of a Data-Center Switch. In *ACM SOSR*, 2015.
- [75] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable Packet Scheduling at Line Rate. In *ACM SIGCOMM*, 2016.
- [76] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster. Composing Dataplane Programs with μ P4. In *ACM SIGCOMM*, 2020.
- [77] R. Stoyanov and N. Zilberman. MTPSA: Multi-Tenant Programmable Switches. In *3rd P4 Workshop in Europe*, 2020.
- [78] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4FPGA: A Rapid Prototyping Framework for P4. In *ACM SOSR*, 2017.
- [79] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. K. Ports, and A. Panda. Multitenancy for Fast and Programmable Networks in the Cloud. In *USENIX HotCloud*, 2020.
- [80] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasecky, A. Krishnamurthy, and A. Chen. Runtime Programmable Switches. In *USENIX NSDI*, 2022.

- [81] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu. Hy-
perV: A High Performance Hypervisor for Virtualization
of the Programmable Data Plane. In *IEEE ICCCN*, 2017.
- [82] P. Zheng, T. Benson, and C. Hu. P4Visor: Lightweight
Virtualization and Composition Primitives for Building
and Testing Modular Programs. In *ACM CoNEXT*, 2018.
- [83] H. Zhu, T. Wang, Y. Hong, D. Ports, A. Sivaraman,
and X. Jin. NetVRM: Virtual Register Memory for
Programmable Networks. In *USENIX NSDI*, 2022.
- [84] N. Zilberman, Y. Audzevich, G. A. Covington, and
A. W. Moore. NetFPGA SUME: Toward 100 Gbps as
Research Commodity. *IEEE Micro*, 2014.

A Daisy-Chain vs. Fully-AXI-L-Based Configuration

As discussed in §3.1, Menshen uses a daisy chain pipeline to configure the Menshen pipeline and uses the AXI-L [28] protocol for safety alone, i.e., to read the reconfiguration packet counter and update the bitmap during reconfiguration. Before using this daisy-chain approach, we considered a different approach based fully on the AXI-L protocol. In this approach, all configuration settings on the FPGA would be set using the AXI-L protocol via PCIe from the host instead of passing a reconfiguration packet through a daisy chain pipeline. We elected to use the daisy-chain approach instead for 2 reasons described below.

First, as one AXI-L write in Corundum can only support a 32-bit data length, we have to write $[625/32] = 20$ and $[205/32] = 7$ times for configuring one entry in the VLIW action table and CAM respectively. For our test modules, we estimate AXI-L reconfiguration time based on the write time of a single AXI-L write. As shown in Figure 12, Menshen’s daisy-chain configuration is much faster than the AXI-L based method, especially for longer entries (i.e., VLIW action table). These benefits are likely to be more pronounced on a larger implementation of Menshen because the entries (both for VLIW action table and CAM) will be even longer in that case. Second, the daisy-chain approach is more similar in style to how programmable switch ASICs are configured today, hence, it is preferable for an eventual ASIC implementation of Menshen.

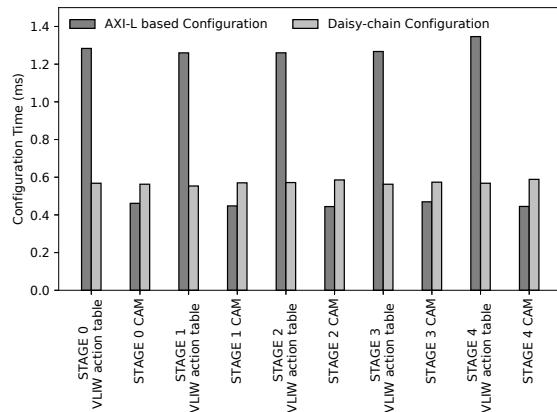


Figure 12: Configuration time comparison for AXI-L based (estimated) and Menshen’s daisy-chain configuration (measured).

B Isolation of ternary match tables using the Xilinx CAM IP

While our current Menshen implementation only supports exact matching, we could reuse our implementation strategy (the Xilinx CAM IP) for ternary matching as well. However, supporting isolation between the ternary match tables of multiple different modules requires some care. This is to

ensure that updates to the ternary match-action rules for one module do not cause updates to the ternary match-action rules for another module.

In the case of ternary matching, the Xilinx CAM IP block uses the address of a CAM entry as the TCAM priority to determine which entry to return when there are multiple matches [31]. Concretely, the Xilinx CAM IP block can prioritize either the entry with the lowest address or the highest address. To support isolation on top of this block, first, we append the module ID (i.e., VLAN ID) to ternary match-action rules as we do currently for exact matches (§3). Second, we allocate contiguous addresses within the Xilinx CAM IP block to a particular module.

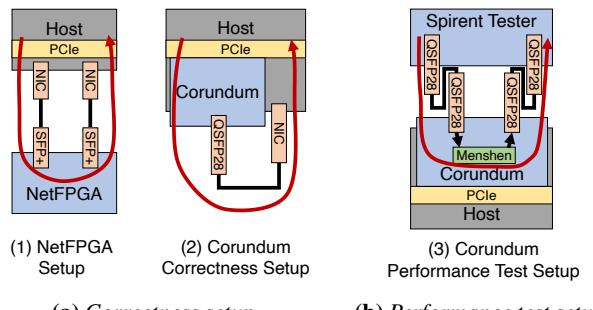
Appending the module ID ensures that a module’s packets do not match any other module’s match-action rules. Allocating contiguous addresses ensures that a new match-action rule can be added (or an old rule can be updated) for a module with disruption to that module’s match-action rules alone—and importantly, without disturbing the rules for any other modules.⁷

C Hardware resources in Menshen

Hardware Resource	Description
Packet Filter	A 32-bit bitmap, and a 4-byte reconfiguration packet counter
PHV	2-byte, 4-byte, 6-byte containers, each type has 8 containers a 32-byte container for platform-specific metadata
Parsing action	16 bits wide
Parser and deparser table	10 parsing actions, 160 bits wide, 32 entries deep
Key extractor table	38 bits wide, 32 entries deep
Key mask table	193 bits wide, 32 entries deep
Exact match table	205 bits wide, 16 entries deep
ALU Action	25 bits wide
VLIW action table	25 ALU actions, 625 bits wide, 16 entries deep
Segment table	16 bits wide, 32 entries deep
Stages	5
Module ID	12 bits

Table 5: Hardware resources in Menshen

D Experimental setup



(a) Correctness setup.

(b) Performance test setup.

Figure 13: Testbed setup. Red arrow shows packet flow.

⁷Note that a new rule can be added to a module only if there are still empty addresses within that module’s chunk of contiguously allocated addresses.

Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks

Yiwen Zhang*, Yue Tan*[◊], Brent Stephens[†], and Mosharaf Chowdhury*

*University of Michigan, [◊]Princeton University, [†]University of Illinois at Chicago

Abstract

Kernel-bypass networking (KBN) is becoming the new norm in modern datacenters. While hardware-based KBN offloads all dataplane tasks to specialized NICs to achieve better latency and CPU efficiency than software-based KBN, it also takes away the operator’s control over network sharing policies. Providing policy support in multi-tenant hardware KBN brings unique challenges – namely, preserving ultra-low latency and low CPU cost, finding a well-defined point of mediation, and rethinking traffic shapers. We present Justitia to address these challenges with three key design aspects: (i) Split Connection with message-level shaping, (ii) sender-based resource mediation together with receiver-side updates, and (iii) passive latency monitoring. Using a latency target as its knob, Justitia enables multi-tenancy policies such as predictable latencies and fair/weighted resource sharing. Our evaluation shows Justitia can effectively isolate latency-sensitive applications at the cost of slightly decreased utilization and ensure that throughput and bandwidth of the rest are not unfairly penalized.

1 Introduction

To deal with the growing demands of ultra-low latency with high throughput (message rates) and high bandwidth in large fan-out services, ranging from parallel lookups in in-memory caches [16, 30, 32] and resource disaggregation [2, 22, 52] to analytics and machine learning [1, 26, 47], kernel-bypass networking (KBN) is becoming the new norm in modern datacenters [14, 23, 43, 44, 64]. As the name suggests, with KBN, applications bypass the operating system (OS) kernel to improve performance while relieving the CPU.

There are two major trends in KBN today. *Software-based KBN* (e.g., DPDK) removes the kernel from the data path and performs packet processing in the user space. In contrast, *hardware-based KBN* (e.g., RDMA) further lowers latency by at least one order of magnitude and reduces CPU usage by offloading dataplane tasks to specialized NICs (e.g., RDMA NICs) with on-board compute.

Hardware KBN, however, takes away the operator’s control over network sharing policies such as prioritization, isolation, and performance guarantees. Unlike software KBN, coexisting applications must rely on the specialized NIC to arbitrate among data transfer operations once they are posted to the hardware. We observe that existing hardware KBNs

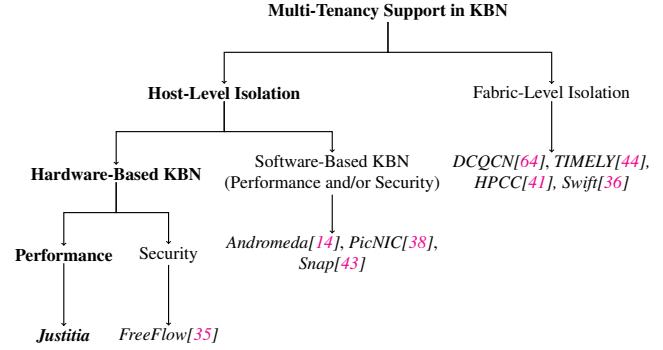


Figure 1: Design space for multi-tenancy support in KBN.

provide poor support for multi-tenancy. For example, even for real-world applications such as DARE [49], eRPC [32], and FaSST [31], sharing the same NIC leads to severe performance anomalies including unpredictable latency, throttled throughput (i.e., lower message rates), and unfair bandwidth sharing (§3). In this paper, we aim to address the following question: *Can we marry the benefits of software KBN with the efficiency of hardware KBN and enable fine-grained multi-tenancy support?*

Recent works have explored multi-tenancy support in large-scale software-based KBN deployments [14, 38, 43]. Their designs enforce fine-grained sharing policies such as performance and security (address space) isolation at the end hosts and pair with fabric-level solutions (e.g., congestion control) in case the network fabric becomes a bottleneck (Figure 1). However, existing software KBN solutions cannot be applied to hardware-based KBN due to three unique challenges:

1. Because host CPU is no longer involved, common CPU-based resource allocation mechanism cannot be applied. Instead, tenants issue RDMA operations with arbitrary data load at no CPU cost, which leaves no obvious point of control to exert resource mediation.
2. Hardware offloading brings packetization from user space into the NIC, disabling fine-grained user-space shaping at the packet level [27, 51].
3. It is also crucial to preserve hardware-based KBN’s efficiency (i.e., single μ s latency and low CPU cost¹) while providing multi-tenancy support.

We present Justitia, a software-only solution that enables

¹This does not apply to applications that aim for low latency or high message rates and busy spin their cores for maximum performance.

multi-tenancy support in hardware-based KBN, to address the aforementioned challenges (§4). Our key idea is to introduce an efficient software mediator in front of the NIC that can implement performance-related multi-tenancy policies – including (1) fair/weighted resource sharing and (2) predictable latencies while maximizing utilization or a mix of the two. Given that RDMA is the primary hardware-based KBN implementation today, in this paper, we specifically focus our solutions on RDMA NICs (RNICs).

Enabling fine-grained sharing policies in RDMA requires an efficient way of managing RNIC resources (i.e., link bandwidth and execution throughput). To this end, we propose *Split Connections* that decouple a tenant application’s intent from its actuation and introduces a point of resource mediation. Justitia mediates RNIC resources by combining the benefits of sender-based and receiver-based design. RDMA operations are split and paced at the sender side before placing them onto the RNIC; receiver-side updates are collected to avoid spurious resource allocation caused by either incast or RDMA READ contention. Shaping is performed at the message level, where message sizes and their pacing rate are adjusted dynamically based on the current policy in use. By splitting RDMA connections, Justitia can effectively manage tenants’ connections to consume RNIC resources based on the policy we set instead of letting tenants themselves compete by arbitrarily issuing RDMA operations.

To provide predictable latencies for latency-sensitive applications, Justitia introduces the concept of *reference flow* and monitors its latency instead of intercepting low-latency tenant applications. By comparing the latency measurements of many reference flows from the same sender machine to different receivers, Justitia can quickly detect (local and remote) RNIC resource contention. Given a tail latency target, Justitia maximizes RNIC resource utilization without violating the target. When the target is unachievable, based on the operator-defined policy, Justitia can choose to ensure that each of the competing n entities gets at least $\frac{1}{n}$ th of one of the RNIC’s two resources, extending the classic hose model of network sharing [17] to multi-resource RNICs.

We have implemented (§5) and evaluated (§6) Justitia on both InfiniBand and RoCEv2 networks. It provides multi-tenancy support among different types of applications without incurring high CPU usage (1 CPU core per host), introducing additional overheads, or modifying application codes. For example, using Justitia, DARE’s tail latency improves by $3.4\times$ when running in parallel with Apache Crail [5, 60], a bandwidth-sensitive storage application, and Justitia preserves 81% of Crail’s original performance. Justitia also complements RDMA congestion control protocols like DCQCN [64] while further mitigating receiver-side RNIC contention, and reduces tail latency even when the network is congested.

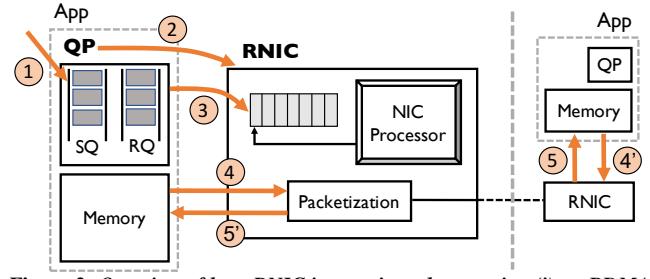


Figure 2: Overview of host-RNIC interaction when posting (i) an RDMA WRITE operation (① → ② → ③ → ④ → ⑤) and (ii) an RDMA READ operation (① → ② → ③ → ④' → ⑤').

2 Background

Recent works [36, 38] have discovered unpredictable latencies due to end-host resource contention, but their primary focus is on receiver-side engine congestion in software-based KBN. In this work, we aim to emphasize that *sender-side resource contention* in hardware-based KBN such as RDMA can also lead to severe performance degradation when multiple tenants coexist. An ideal solution should address both sender- and receiver-side issues. In this section, we give an overview on how an RDMA operation is performed, followed by the root cause of RDMA’s lack of multi-tenancy support.

2.1 Life Cycle of an RDMA Operation

RDMA enables direct access between user-registered memory regions without involving the OS kernel, offloading data transfer tasks to the RNIC. Applications initiate RDMA operations by posting Work Requests (WRs) via Queue Pairs (QPs) to describe the messages to transmit. Figure 2 shows how an RDMA application interacts with an RNIC to initiate an RDMA operation. To start an RDMA WRITE, ① the user application places a Work Queue Element (WQE) describing the message to the Send Queue (SQ), and ② rings a doorbell to notify the RNIC by writing its QP number into the corresponding doorbell register on RNIC. At this point, the user application has completed its task and offloads the rest of the work to RNIC. After the RNIC gets notified, it ③ fetches and processes the requests from the send queue, and ④ pulls the message from user memory, splits it into packets, and sends it to the remote RNIC. Finally, the remote RNIC ⑤ writes the received message directly into the remote memory.

In the case of an RDMA READ operation, the user application again posts the WQE and notifies the RNIC to collect it (① → ③). The local RNIC then ④ notifies the remote RNIC to pull the data from remote memory, and ⑤ places the message back to local memory after de-packetizing the received packets. Despite the opposite direction of data transfer, the remote OS remains passive just as the case with an RDMA WRITE. In both cases, the sender of the RDMA operation actively controls what goes into the RNIC while the remote side stays passively unaware.²

²This is true even for two-sided operations that require the receiver to post WQEs to its Receive Queue before a Send Request arrives. We still

2.2 Lack of Multi-Tenancy Support

RDMA lacks multi-tenancy support for two primary reasons: (i) tenants/applications compete for multiple RNIC resources, and (ii) RNIC processes ready-to-consume message in a greedy fashion to maximize utilization. Both are related to different symptoms of the isolation issues.

Multi-Resource Contention There exist two primary resources that need to be shared on an RNIC: *link bandwidth* and *execution throughput*. Bandwidth-sensitive applications consume RNIC’s link bandwidth to issue large DMA requests. Throughput-sensitive applications, on the other hand, consume RNIC’s execution throughput to issue small DMA requests in batches. Latency-sensitive applications, however, consume neither resource with the small messages they sparsely send. As we will soon show (§3), isolation anomalies can occur when applications compete for different resources.

Greedy Processing for High Utilization Although the actual RNIC implementation details are private, we can consider two hypotheses on how RNIC handles multiple requests simultaneously: either the RNIC buffers WQEs collected in ③ in Figure 2 from multiple applications and arbitrates among them using some scheduling mechanism; or it processes them in a greedy manner. When a latency-sensitive application competes with a bandwidth-sensitive application, too much arbitration in the former can cause low resource utilization (e.g., unable to catch up the line rate), whereas too little arbitration in the latter leads to head-of-line (HOL) blocking (which leads to latency variation). Our observations across all three RDMA implementations (§3), where applications using small messages are consistently affected by the ones using larger ones, suggest the latter. Note that even though receiver-side congestion can also happen during step ⑤ as pointed out in [38], both root causes can easily stem from the sender side of the operation via step ③ and thus cannot be ignored. We elaborate on how Justitia mitigates both sender- and receiver-side issues in Section 4.

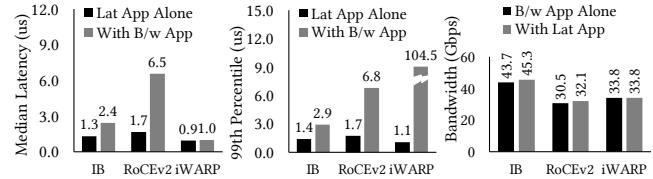
3 Performance Isolation Anomalies in RDMA

This section establishes a baseline understanding of sharing characteristics in hardware KBN and identifies common isolation anomalies across different RDMA implementations with both microbenchmarks (§3.1) and highly optimized, state-of-the-art RDMA-based applications (§3.2).

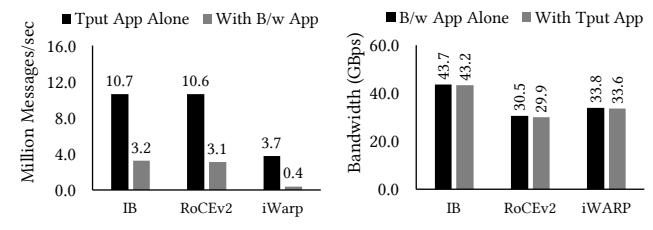
To study RDMA sharing characteristics among applications with different objectives, we consider three major types of RDMA-enabled applications:

1. *Latency-Sensitive*: Sends small messages and cares about the individual message latencies.
2. *Throughput-Sensitive*: Sends small messages in batches to maximize the number of messages sent per second.

consider the receiver as passive because it can only control where to place a message but cannot control *when* a message will arrive.



(a) *Latency App (Med)* (b) *Latency App (99th)* (c) *Bandwidth App*
Figure 3: *Latency-sensitive applications require isolation against bandwidth-sensitive applications.*



(a) *Throughput App* (b) *Bandwidth App*
Figure 4: *Throughput-sensitive application requires isolation from bandwidth-sensitive applications.*

3. *Bandwidth-Sensitive*: Sends large messages with high bandwidth requirements.

Summary of Key Findings:

- Both latency- and throughput-sensitive applications need isolation from bandwidth-sensitive applications (§3.1.1).
- If only latency- or throughput-sensitive applications (or a mix of the two types) compete, they are isolated from each other (§3.1.2).
- Multiple bandwidth-sensitive applications can lead to unfair bandwidth allocations depending on their message sizes (§3.1.3).
- Highly optimized, state-of-the-art RDMA-based systems also suffer from the anomalies we discovered (§3.2).

In the rest of this section, we describe our experimental settings and elaborate on these findings.

3.1 Observations From Microbenchmarks

We performed microbenchmarks between two machines with the same type of RNIC, where both are connected to the same RDMA-enabled switch. For most of the experiments, we used 56 Gbps Mellanox ConnectX-3 Pro for InfiniBand, 40 Gbps Mellanox ConnectX-4 for RoCEv2, and 40 Gbps Chelsio T62100 for iWARP; 10 and 100 Gbps settings are described similar. More details on our hardware setups are in Table 1 of Appendix A.

Our benchmarking applications are written based on Mellanox perfest [61] and each of them uses a single Queue Pair. Unless otherwise specified, latency-sensitive applications in our microbenchmarks send a continuous stream of 16B messages, throughput-sensitive ones send a continuous stream of batches with each batch having 64 16B messages, and bandwidth-sensitive applications send a continuous stream of 1MB messages. Although all applications send messages

using RDMA WRITES over reliable connection (RC) QPs in the observations below, other verbs show similar anomalies as well. We defer the usage and discussion of hardware virtual lanes to Section 6.3.

3.1.1 Both Latency- and Throughput-Sensitive Applications Require Isolation

The performance of the latency-sensitive applications deteriorate for all RDMA implementations (Figure 3). Out of the three implementations we benchmarked, InfiniBand and RoCEv2 observes $1.85\times$ and $3.82\times$ degradations in median latency and $2.23\times$ and $4\times$ at the 99th percentile. While iWARP performs well in terms of median latency, its tail latency degrades dramatically ($95\times$).

Throughput-sensitive applications also suffer. When a background bandwidth-sensitive application is running, the throughput-sensitive ones observe a throughput drop of $2.85\times$ or more across all RDMA implementations (Figure 4). Note that in our microbenchmark with 1 QP per application, throughput-sensitive applications that consume NIC execution throughput hit the bottleneck. This does not imply RNIC always favors link bandwidth over execution throughput. We notice RNIC bandwidth starts to become the bottleneck when there exists $4\times$ more throughput-sensitive applications.

More importantly, both latency- and throughput-sensitive applications experience more severe performance degradations (e.g., 139X worse latency with the presence of 16 bandwidth applications) as more bandwidth-sensitive applications join the competition, which is prevalent in shared datacenters [23, 64]. Appendix B.1 provides more details.

3.1.2 Latency-Sensitive Applications Coexist Well; So Do Throughput-Sensitive Ones

We observe no obvious anomalies among latency- or throughput-sensitive applications, or a mix of the two types. Detailed results can be found in Appendix B.

3.1.3 Bandwidth-Sensitive Applications Hurt Each Other

Unlike latency- and throughput-sensitive applications, bandwidth-sensitive applications with different message sizes do affect each other. Figure 5 shows that a bandwidth-sensitive application using 1MB messages receive smaller share than one using 1GB messages. The latter receives $1.42\times$, $1.22\times$ and $1.51\times$ more bandwidth in InfiniBand, RoCEv2, and iWARP, respectively.

3.1.4 Anomalies are Present in Faster Networks Too

We performed the same benchmarks on 100 Gbps InfiniBand, only to observe that most of the aforementioned anomalies are still present. Appendix C.1 has the details.

3.2 Isolation Among Real-World Applications

In this section, we demonstrate how real RDMA-based systems fail to preserve their performance in the presence of the

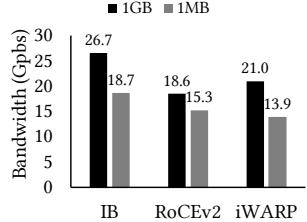


Figure 5: Anomalies among Bandwidth-sensitive applications

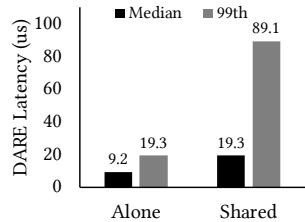


Figure 6: Latency of DARE’s Put Bandwidth-sensitive applications and Get operations when coexisting with Apache Crail’s storage traffic.

aforementioned anomalies.

Specifically, we performed experiments with Apache Crail [5, 60] and DARE [49]. Crail is a bandwidth-hungry distributed data storage system that utilizes RDMA. In contrast, DARE is a latency-sensitive system that provides high-performance replicated state machines through the use of a strongly consistent RDMA-based key-value store.

In these experiments, we deployed DARE in a cluster of 4 nodes with 56 Gbps Mellanox ConnectX-3 Pro NIC on InfiniBand with 64GB memory. Crail is deployed in the same cluster with one node running the namenode and one other node running the datanode.

To evaluate the performance of Crail, we launch 8 parallel writes (each to a different file) in Crail’s data storage with the chunk size of the data transfer configured to be 1MB, and we measure the application-level throughput reported by Crail. To evaluate the performance of DARE, one DARE client running on the same server as the namenode of Crail issues PUT and GET operations (each PUT is followed by a GET) to the DARE server on the other 3 nodes with a sweep of message sizes from 8 byte to 1024 bytes, and we measure the application-level latency reported by DARE.

Figure 6 plots the latency of DARE’s queries with and without the presence of Crail. In this experiment, we observe a $4.6\times$ increase in DARE’s tail latency. Additionally, regardless of whether it is competing with DARE, Crail’s total write throughput stays at 51.1 Gbps.

Besides DARE, highly-optimized RDMA-based RPC system such as FaSST [31] and eRPC [32] also suffer from isolation anomalies caused by unmanaged resource contention on RNICs. In fact, when background bandwidth-heavy traffic is present, FaSST’s throughput experiences a 74% drop (Figure 32) and eRPC’s tail latency increases by $40\times$ (Figure 33). More details can be found in Appendix C.2.

3.3 Congestion Control is not Sufficient

To demonstrate that DCQCN [64] and PFC are not sufficient to solve these anomalies, we performed the benchmarks again with PFC enabled at both the NICs and switch ports, DCQCN [64] enabled at the NICs, and ECN markings enabled on a Dell 10 Gbps Ethernet switch (S4048-ON). In these experiments, latency- and throughput-sensitive applications still suffer unpredictably (Section 6.3 has detailed results). This is because DCQCN focuses on fabric-level isolation whereas

the observed anomalies happen at the end host due to RNIC resource contention (§2.2).

4 Justitia

Justitia enables multi-tenancy in hardware-based KBN, with a specific focus on enabling two performance-related policies: (1) fair/weighted resource sharing, or (2) predictable latencies while maximizing utilization, or a mix of the two. Note that we restrict our focus on a *cooperative* datacenter environment in this paper and defer strategyproofness [20, 21, 50] to mitigate adversarial/malicious behavior to future work.

Granularity of Control: We define a flow to be a stream of RDMA messages between two RDMA QPs. Justitia can be configured to work either at the flow granularity or at the application granularity by considering all flows between two applications as a whole.³ In this paper, by default, we set Justitia’s granularity of control to be at the application level to focus on application-level performance.

4.1 Key Design Ideas

Justitia resolves the unique challenges of enabling multi-tenancy in hardware KBN with five key design ideas.

- *Tenant-/application-level connection management:* To prevent tenants from hogging RNIC resources by issuing arbitrarily large messages or creating a large number of active QPs at no cost, Justitia provides a tenant-level connection management scheme by adding a shim layer between tenant applications and the RNIC. Tenant operations are handled by Justitia before arriving at the RNIC.
- *Sender-based proactive resource mediation:* Justitia proactively controls RNIC resource utilization at the sender side. This is based on the observation that the sender of an RDMA operation – that decides when an operation gets initiated, how large the message is, and in which direction the message flows – has active control over every aspect of the transmission while the other side of the connection remains passive. Such sender-based control can react before the RNIC takes over and maintain isolation by directly controlling RNIC resources.
- *Dynamic receiver-side updates:* Pure sender-based approaches can sometimes lead to spurious resource allocation when multiple senders coexist but are unaware of each other. Justitia leverage receiver-side updates to provide information (e.g., the arrival or departure of an application) back to the senders to react correctly when a change in the setting happens.
- *Passive latency monitoring:* Instead of actively measuring each application’s latency, which can introduce high overhead, Justitia uses passive latency monitoring by issuing *reference flows* to detect RNIC resource contention.

³Each granularity has its pros and cons when it comes to performance isolation, without any conclusive answer on the right one [46].

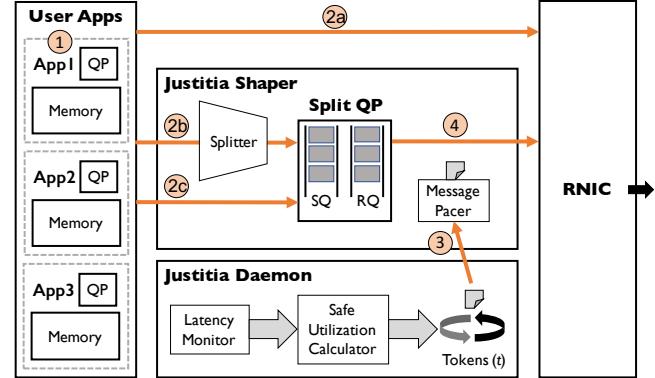


Figure 7: Justitia architecture. Bandwidth- and throughput-sensitive applications are shaped by tokens generated at a regular interval by Justitia. Latency-sensitive ones are not paced at all.

- *Message-level shaping with splitting:* Justitia performs shaping at the message level to suit RDMA’s message-oriented transport layer. At the message level, it is easy to apply specific strategies to control how messages enter the RNIC based on their sizes and the resource they consume. Large messages are split into roughly equal-sized sub-messages or chunks to (i) avoid a single message requesting too many RNIC resources; (ii) facilitate network sharing policies such as fair/weighted bandwidth share; and (iii) mitigate HOL Blocking for latency-sensitive applications.

4.2 System Overview

Figure 7 presents a high-level system overview of Justitia handling an RDMA WRITE operation (to compare with Figure 2). Each machine has a Justitia daemon that performs latency monitoring and proactive rate management, and applications create QPs using the existing API to perform RDMA communication. Justitia relies on applications to optionally identify their application type. By default, they are treated as bandwidth-sensitive. VMs, containers, bare-metal applications, and SR-IOV are all compatible with the design of Justitia.

As before, the user application starts an RDMA WRITE operation by ① posting a WQE into the Send Queue. Latency-sensitive applications will ②a bypass Justitia and directly interact with the RNIC as shown in Figure 2. The other two types of applications will enter Justitia’s shaper. The Splitter will ②b split the big message from a bandwidth-sensitive applications equally into sub-messages or ②c do nothing given a small message from a throughput-sensitive application. We introduce *Split Connection* – and corresponding split queue pair (Split QP) – to handle the messages passed through the Splitter. Before sending out the message, it ③ asks the daemon to fetch a token from Justitia, which is generated at a rate to maximize RNIC resource utilization consumed by resource-hungry applications. Once the token is fetched, the Split QP ④ posts a WQE for the sub-message into its SQ and rings the door bell to notify the RNIC. The RNIC then grabs

the WQE from Split QP, issue a DMA read for the actual data in application’s memory region, and sends the message to the remote side (arrows not shown in the figure). Steps ③ and ④ repeat until all messages in the Split QP have been processed. The implementation details of Split QP is in Section 5.1.

The Justitia daemon in Figure 7 is a background process that performs latency monitoring and proactive rate management to maximize RNIC resource utilization when latency target is met.

4.3 Justitia Daemon

Justitia daemon performs two major tasks: (i) proactively manages rate of all bandwidth- and throughput-sensitive applications using the hose model [17]; (ii) ensures predictable performance for latency-sensitive applications while maximizing RNIC resource usage.

4.3.1 Minimum Guaranteed Rate

Justitia enforces rate based on the classic hose model [17], and always maintains a minimum guaranteed rate R_{min} :

$$R_{min} = \frac{\sum w_B^i + \sum w_T^i}{\sum w_B^i + \sum w_T^i + \sum w_L^i} \times MaxRate$$

where w_X^i represents the weight of application i of type X (i.e., bandwidth-, throughput-, or latency-sensitive), and $MaxRate$ represents the maximum RNIC bandwidth or maximum RNIC throughput (both are pre-determined on a per-RNIC basis) depending on the type of the application. The idea of R_{min} is to recognize the existence of latency-sensitive applications, and provide isolation for them by *taking out their share from the RNIC resources which otherwise they cannot acquire by themselves*. In the absence of latency-sensitive applications (i.e., $\sum w_L^i = 0$), R_{min} is equivalent to $MaxRate$, and all the resource-hungry applications share the entire RNIC resources. If all applications have equal weights, and there exist B bandwidth-, T throughput-, and L latency-sensitive applications, R_{min} can be simplified as $\frac{B+T}{B+T+L} \times MaxRate$.

In the presence of a large number of latency-sensitive applications, R_{min} could be really small, essentially removing RNIC resource guarantee. To accommodate such cases, one can fix $L = 1$ no matter how many latency-sensitive applications join the system since they do not consume much of RNIC’s resources. We find this setting works well in practice (§6.4) and make it the default option for Justitia.

With R_{min} provided, Justitia then *maximizes RNIC’s safe resource utilization* (which we denote $SafeUtil$) until the performance of latency-sensitive applications crosses the target tail latency ($Target_{99}$).

4.3.2 Latency Monitoring via Reference Flows

Justitia does not interrupt or interact with latency-sensitive applications because (i) they cannot saturate either of the two RNIC resources, and (ii) interrupting them fails to preserve RDMA’s ultra-low latency.

Pseudocode 1 Maximize SafeUtil

```

1: procedure ONLATENCYFLOWUPDATE( $L$ ,  $Estimated_{99}$ )
2:   if  $L = 0$  then       $\triangleright$  Reset if no latency-sensitive applications
3:      $SafeUtil = MaxRate$ 
4:   else
5:     if  $Estimated_{99} > Target_{99}$  then
6:        $SafeUtil = \max(\frac{SafeUtil}{2}, R_{min})$ 
7:     else
8:        $SafeUtil = SafeUtil + 1$ 
9:     end if
10:   end if
11:    $\tau = Token_{Bytes}/ SafeUtil$ 
12: end procedure
```

Instead, whenever there exists one or more latency-sensitive applications to particular receiving machine, Justitia maintains a *reference flow* to that machine which keeps sending 10B messages to the same receiver as the latency-sensitive applications in periodic intervals (by default, $RefPeriod = 20 \mu s$) to estimate the 99th percentile ($Estimated_{99}$) latency for small messages. By monitoring its own reference flow, Justitia does not need to wait on latency-sensitive applications to send a large enough number of sample messages for accurate tail latency estimation. It does not add additional delay by directly probing those applications either.

Given the stream of measurements, Justitia maintains a sliding window of the most recent $RefCount$ (=10000) measurements for a reference flow estimate its tail latency.

4.3.3 Maximizing SafeUtil

Using the selected latency measurement from the reference flow(s), Justitia maximizes $SafeUtil$ based on the algorithm shown in Pseudocode 1. To continuously update $SafeUtil$, Justitia uses a simple AIMD scheme that reacts to $Estimated_{99}$ every $RefPeriod$ interval as follows. If the estimation is above $Target_{99}$, Justitia decreases $SafeUtil$ by half; $SafeUtil$ is guaranteed to be at least R_{min} . If the estimation is below $Target_{99}$, Justitia slowly increases $SafeUtil$. Because $SafeUtil$ ranges between R_{min} to the total RNIC resources and latency-sensitive applications are highly sensitive to too high a utilization level, our conservative AIMD scheme, which drops utilization quickly to meet $Target_{99}$, works well in practice.

To determine the value of $Target_{99}$, we constructs a latency oracle that performs pair-wise latency measurement by issuing reference flows across all the nodes in the cluster when there is no other background. Microsoft applies a similar approach in [24], which is shown to work well in estimating steady-state latency in the cluster. We adopt this approach to give a good estimate of the latency target under well-isolated scenarios.

4.3.4 Token Generation And Distribution

Justitia uses multi-resource tokens to enforce $SafeUtil$ among the B bandwidth- and T throughput-sensitive applications in a fair or weighted-fair manner. Each token represents a fixed

amount of bytes ($Token_{Bytes}$) and a fixed number of messages ($Token_{Ops}$). In other words, the size of $Token_{Bytes}$ determines the chunk size a message from bandwidth-sensitive application is split into. A token is generated every τ interval, where the value of τ depends on $SafeUtil$ as well as on the size of each token. For example, given 48 Gbps application-level bandwidth and 30 Million operations/sec on a 56 Gbps RNIC, if $Token_{Bytes}$ is set to 1MB, then we set $Token_{Ops} = 5000$ ops and $\tau = 167 \mu s$.

Justitia daemon continuously generates one token every τ interval and distributes it among the active resource-hungry applications in a round-robin fashion based on application weights w_X^i . When $w_X^i = 1$ for all applications, Justitia enforces traditional max-min fairness; otherwise, it enforces weighted fairness. Each application independently enforces its rate using one of the shapers described below.

4.4 Justitia Shapers

Justitia shapers – implemented in the RDMA driver – enforce utilization limits provided by the Justitia daemon-calculated tokens. There are two shapers in Justitia: one for bandwidth- and another for throughput-sensitive applications.

Split Connection Justitia introduces the concept of a Split Connection to provide an interface to coordinate between tenant applications and the RNIC. It consists of a message splitter and custom *Split QPs* (§5.1) to initiate RDMA operations for tenants. Each application’s Split Connection cooperate with Justitia daemon to pace split messages transparently.

Shaping Bandwidth-Sensitive Applications. This involves two steps: *splitting* and *pacing*. For any bandwidth-sensitive application, Justitia *transparently* divides any message larger than $Token_{Bytes}$ into $Token_{Bytes}$ -sized chunks to ensure that the RNIC only sees roughly equal-sized messages. Splitting messages for diverse RDMA verbs – e.g., one-sided vs. two-sided – requires careful designing (§5.1).

Given chunk(s) to send, the pacer requests for token(s) from the Justitia daemon by marking itself as an active application. Upon receiving a token, it transfers chunk(s) until that token is exhausted and repeats until there is nothing left to send. The application is notified of the completion of a message only after all of its split messages have been transferred.

Batch Pacing for Throughput-Sensitive Applications. These applications typically deal with (batches of) small messages. Although there is no need for message splitting, pacing individual small messages requires the daemon to generate and distribute a large number of tokens, which can be CPU-intensive. Moreover, for messages as small as 16B, such fine-grained pacing cannot preserve RDMA’s high message rates.

To address this, Justitia performs *batch pacing* enabled by Justitia’s multi-resource token. Each token grants an application a fixed batch size ($Token_{Ops}$) that it can send together before receiving the next token. Batch pacing on throughput-sensitive applications removes the bottleneck on token generation.

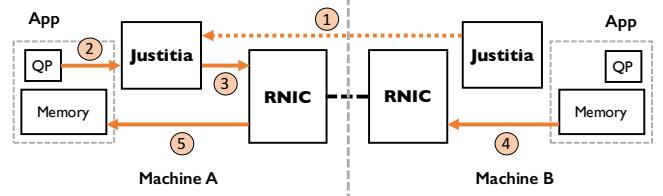


Figure 8: How Justitia handles READS via remote control.

action and distribution; it also relieves daemon CPU cost with a unified token bucket.

Mitigating Head-of-Line Blocking. One of the foremost goals of Justitia is to mitigate HOL blocking caused by the bandwidth-sensitive applications to provide predictable latencies. To achieve this goal, we need to split messages into smaller chunks and pace them at a certain rate (enforcing $SafeUtil$) with enough spacing between them to minimize the blocking. However, this simple approach creates a dilemma. On the one hand, too large a chunk may not resolve HOL Blocking. On the other hand, too small a chunk may not be able to reach $SafeUtil$. It also leads to increased CPU overhead from using a spin loop to fetch tokens generated in a very short period in which context switches are not affordable. This is a manifestation of the classic performance isolation-utilization tradeoff. We discuss how to pick the chunk size in Section 5.2.

4.5 Dynamic Receiver-Side Updates

Justitia relies on receiver-side updates to coordinate among multiple senders to avoid spurious allocation of RNIC resources. The benefits of this design is three-fold: (i) it coordinates with multiple senders to provide the correct resource allocation; (ii) it keeps track of RDMA READ issued which can collide with applications issuing RDMA WRITE in the opposite direction; (iii) it mitigates receiver-side engine congestion by rate-limiting senders with the correct fan-in information.

The updates are communicated among Justitia Daemons only when a change in the application state happened to a certain receiver (i.e., an arrival or an exit of an application) is detected. Two-sided operations, SEND and RECV, are selected in such case so that the daemon gets notified when an update arrives. Once a change is detected by a sender, it informs the receiver, which then broadcasts the change back to all the senders it connects to so that they can update the correct R_{min} . In such case, R_{min} considers remote resource-hungry application count as part of the total share. If the local daemon has not issued a reference flow and a remote latency-sensitive applications launches to the receiver, the daemon will start a new reference flow to start latency monitoring.

Handling READS RDMA specification allows remote machines to read from a local machine using the RDMA READ verb. RDMA READ operations issued by machine A to read data from machine B compete with all sending operations (e.g., RDMA WRITE) from machine B. Consequently, Justitia must handle remote READs as well.

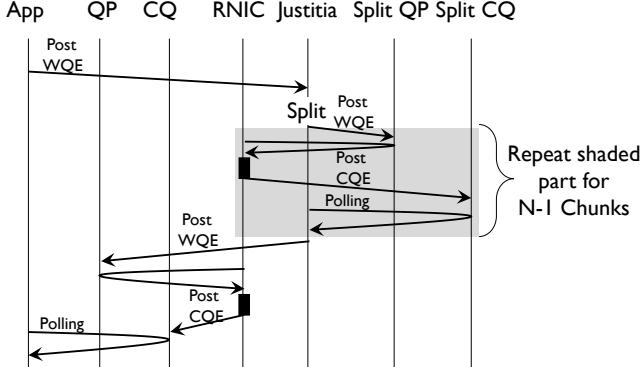


Figure 9: High-level overview of transparent message splitting in Justitia for one-sided verbs using Split QP. Times are not drawn to scale. Two-sided verbs involve extra bookkeeping.

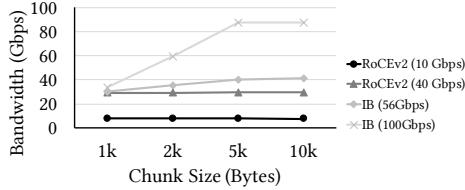


Figure 10: Maximum achievable bandwidth vs. chunk sizes.

In such a case, the receiver of the READ operation, machine B , sends the updated guaranteed utilization R_{min} , with the updated count of senders including remote READ applications) as shown in ① in Figure 8. After A receives that utilization, it operates RDMA READ by interact with Justitia normally via ② → ⑤ and enforces the updated rate.

5 Implementation

We have implemented the Justitia daemon as a user-space process in 3,100 lines of C, and the shapers are implemented inside individual RDMA drivers with 5,200 lines of C code. Our current implementation focuses on container/bare-metal applications. Justitia code is available at <https://github.com/SymbioticLab/Justitia>.

5.1 Transparently Splitting RDMA Messages

Justitia splitter transparently divides large messages of bandwidth-sensitive applications into smaller chunks for packing. Our splitter uses a custom QP called a *Split QP* to handle message splitting, which is created when the original QP of a bandwidth-sensitive flow is created. A corresponding *Split CQ* is used to handle completion notifications. A custom completion channel is used to poll those notifications in an event-triggered fashion to preserve low CPU overhead.

To handle one-sided RDMA operations, when detecting a message larger than $Token_{Bytes}$, we divide the original message into chunks and only post the last chunk to the application's QP (Figure 9). The rest of the chunks are posted to the Split QP. Split QP ensures all chunks have been successfully transferred before the last chunk handled by the application's QP. The two-sided RDMA operations such as SEND are handled in a similar way, with additional flow control messages for the chunk size change and receive requests

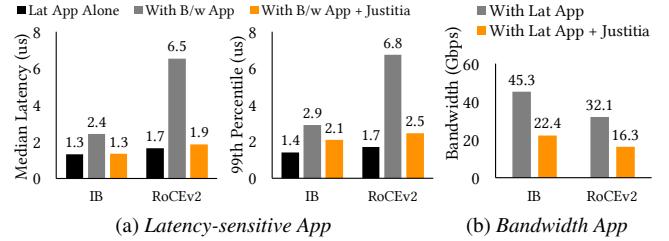


Figure 11: Performance isolation of a latency-sensitive application running against a bandwidth-sensitive one.

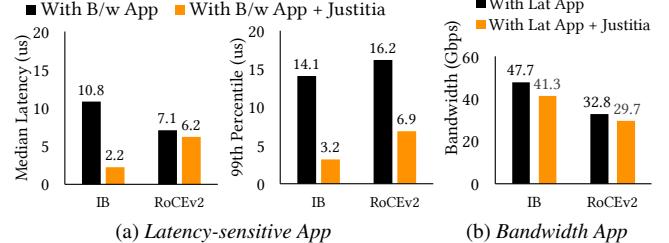


Figure 12: Latency of a latency-sensitive application running against a bandwidth-sensitive application with 4 QPs with a relaxed latency target (10 μ s).

to be pre-posted at the receiver side.

5.2 Determining Token Size for Bandwidth Target

One of the key steps in determining *SafeUtil* is deciding the size of each token. Because the RNIC can become throughput-bound for smaller messages instead of bandwidth-bound, we cannot use arbitrarily small messages to resolve HOL blocking. At the same time, given a utilization target, we want to use the smallest *TokenBytes* value to achieve that target to reduce HOL blocking while maximizing utilization.

Instead of dynamically determining it using another AIMD-like process, we observe that (i) this is an RNIC-specific characteristic and (ii) the number of RNIC types is small. With that in mind, we maintain a pre-populated dictionary to store the smallest token size that can saturate a given rate (to enforce *SafeUtil*) when sending in a paced batch for different latency targets; Justitia simply uses the mappings during runtime. When latency-sensitive applications are not present, a large token size (1MB) is used. Otherwise, Justitia looks up the token size in the dictionary based on the current *SafeUtil* value. This works well since the lower the *SafeUtil* is, the smaller the chunk size it requires to achieve such *SafeUtil*, and the better it helps mitigating HOL blocking. Based on our microbenchmarks (Figure 10), we pick 5KB as the chunk size when latency-sensitive applications are present.

6 Evaluation

In this section, we evaluate Justitia's effectiveness in providing multi-tenancy support among latency-, throughput-, and bandwidth-sensitive applications on InfiniBand and RoCEv2. To measure latency, we perform 5 consecutive runs and present their median. We do not show error bars when they are too close to the median.

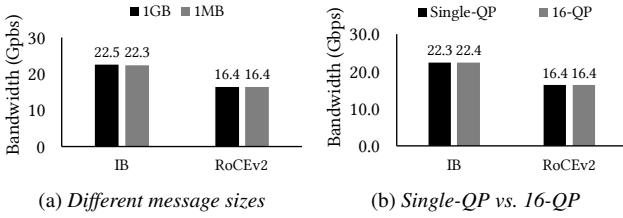


Figure 13: Fair bandwidth share of bandwidth-sensitive applications. (a) different message sizes. (b) different number of QPs.

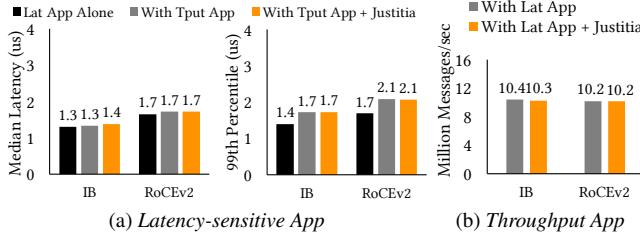


Figure 14: Performance isolation of a latency-sensitive application running against a throughput-sensitive application.

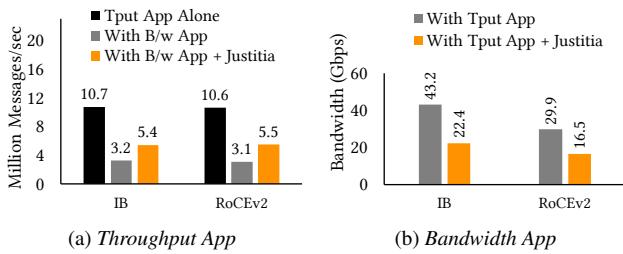


Figure 15: Performance isolation of a throughput-sensitive application running against a bandwidth-sensitive application.

Our key findings can be summarized as follows:

- Justitia can effectively provide multi-tenancy support highlighted in Section 3 both in microbenchmarks and at the application-level (§6.1).
- Justitia scales well to a large number of applications and works for a variety of settings (§6.2); it complements DCQCN and hardware virtual lanes (§6.3).
- Justitia's benefits hold with many latency- and bandwidth-sensitive applications (§6.4), in incast scenarios (§6.5), and under unexpected network congestion (§6.6).

A detailed sensitivity analysis of Justitia parameters can be found in Appendix D.

6.1 Providing Multi-Tenancy Support

We start by revisiting the scenarios from Section 3 to evaluate how Justitia enables sharing policies among different RDMA applications. We use the same setups as those in Section 3. Unless otherwise specified, we set $Target_{99} = 2 \mu\text{s}$ on both InfiniBand and RoCEv2 for the latency-sensitive applications. Justitia works well in 100 Gbps networks too (Appendix C.1). Unless otherwise specified, R_{min} with all applications sharing the same weights is enforced as a default policy.

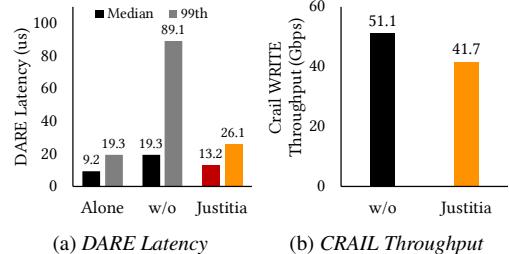


Figure 16: [InfiniBand] Performance isolation of DARE running against Crail.

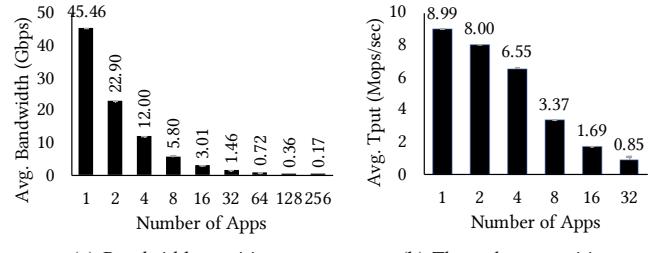


Figure 17: [InfiniBand] Justitia scales to a large number of applications and still provides equal share. The error bars represent the minimum and the maximum values across all the applications.

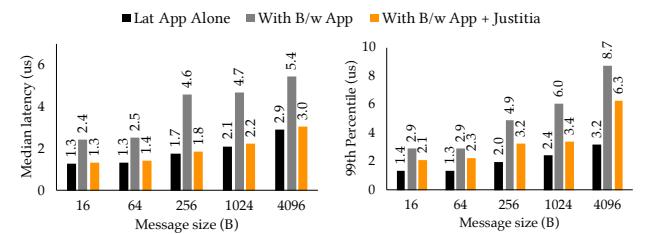


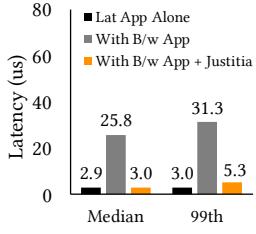
Figure 18: [InfiniBand] Latency-sensitive applications with different message sizes competing against a bandwidth-sensitive app.

Predictable Latency Latency-sensitive applications are affected the most when they compete with a bandwidth-sensitive application. In the presence of Justitia, both median and tail latencies improve significantly in both InfiniBand and RoCEv2 (Figure 11a). Due to the enforcement of R_{min} , the bandwidth-sensitive application is receiving half of the capacity (Figure 11b).

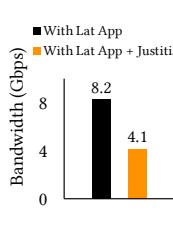
Next we evaluate how Justitia performs when the latency target is set to a relaxed value ($Target_{99} = 10 \mu\text{s}$) that can be easily met (Figure 12). For a slightly high $Target_{99}$, Justitia maximizes utilization, illustrating that splitting and pacing are indeed beneficial.

Fair Bandwidth and Throughput Sharing Justitia ensures that bandwidth-sensitive applications receive equal shares regardless of their message sizes and number of QPs in use (Figure 13) with small bandwidth overhead (less than 6% on InfiniBand and 2% on RoCEv2). The overhead becomes negligible when applying Justitia to throughput- or latency-sensitive applications (Figure 14).

Justitia's benefits extends to the bandwidth- vs throughput-sensitive application scenario as well. In this case, it ensures

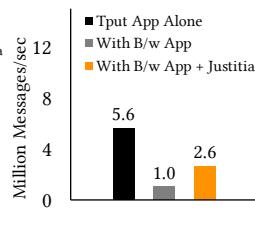


(a) Latency-sensitive App

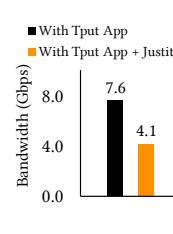


(b) Bandwidth App

Figure 19: [DCQCN] Latency-sensitive application against a bandwidth-sensitive one.

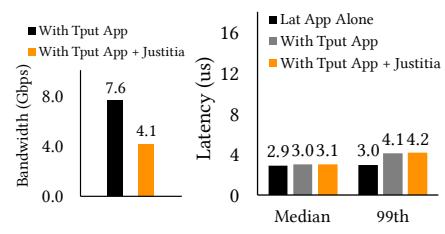


(a) Tput App

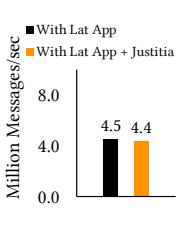


(b) Bandwidth App

Figure 20: [DCQCN] Throughput-sensitive app against a bandwidth-sensitive one.



(a) Latency-sensitive App



(b) Tput App

Figure 21: [DCQCN] Latency-sensitive application against a throughput-sensitive one.

that both receive roughly half of their resources. Figure 15 illustrates this behavior. In both InfiniBand and RoCEv2, the throughput-sensitive application is able to achieve half of its original message rate of itself running alone (Figure 15a). The bandwidth-sensitive application, on the other hand, is limited to half its original bandwidth as expected (Figure 15b).

Justitia and Real-World RDMA Applications To demonstrate that Justitia can isolate highly optimized real-world applications, we performed experiments with DARE and Crail. Thanks to Justitia’s high transparency, we did not need to make any source code changes in Crail (given it is bandwidth-sensitive by default), and we only changed DARE by marking it as latency-sensitive.

From these experiments, we find that Justitia improves isolation for latency-sensitive applications while also preserving high bandwidth of the background storage application. Figure 16 plots the performance of DARE and Crail after applying Justitia with the same setting as in Section 3.2. We observe that, with Justitia, DARE achieves performance that is close to running in isolation even when running alongside Crail, and Justitia improves DARE’s tail latency performance by $3.4\times$ when compared to the baseline scenario while Crail also achieves 81% of its original throughput performance. This is close to the expected throughput of $\frac{8}{9}$ of Crail’s original throughput since in this experiment Justitia treats the 8 parallel writes on top of Crail as separate applications.

Justitia improves performance isolation of FaSST by $2.5\times$ in throughput and eRPC by $32.2\times$ in tail latency. More details can be found in Appendix C.2.

6.2 Justitia Deep Dive

Scalability and Rate Conformance Figure 17a shows that as the number of bandwidth-sensitive applications increases, all applications receive the same amount of bandwidth using Justitia with total bandwidth close to the line rate. Justitia also ensures that all throughput-sensitive application send roughly equal number of messages (Figure 17b).

CPU and Memory Consumption Justitia daemon uses one dedicated CPU core per node to generate and distribute tokens. Its memory footprint is not significant.

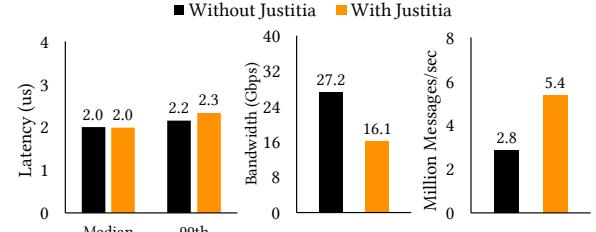


Figure 22: [RoCEv2] A bandwidth-, throughput-, and latency-sensitive application running on two hardware priority queues at the NIC. The latency-sensitive application uses one queue, while the other two share the other queue.

Varying Message Sizes Justitia can provide isolation at a wide range of message sizes for latency-sensitive applications (Figure 18). The bandwidth-sensitive application receives half the bandwidth in all cases.

6.3 Justitia + X

Justitia + DCQCN The anomalies we discover in this paper does not stem from the network congestion, but rather happens at the end hosts. We found that DCQCN falls short for latency- and throughput-sensitive applications (Figures 19, 20, 21). Justitia can complement DCQCN and improve latencies by up to $8.6\times$ and throughput by $2.6\times$.

Justitia + Hardware Virtual Lanes Although RDMA standards support up to 15 virtual lanes [7] for separating traffic classes, they only map to very few hardware shapers and/or priority queues (2 queues in our RoCE NIC) that are rarely sufficient in shared environments [3, 37]. Besides, the hardware rate limiters in the RNIC are slow when setting new rates (2 milliseconds in our setup), making it hard to use with real dynamic arrangement. Moreover, it is desirable to achieve isolation *within each priority queue*, as those hardware resources are often used to provide different levels of quality of service, within which many applications reside.

In this experiment, we show how limited number of hardware queues are insufficient to provide isolation and how Justitia can help in this scenario. we run three applications, one each for each of the three types (Figure 22). Although the latency-sensitive application remains isolated in its own class, the bandwidth- and throughput-sensitive applications compete in the same class. As a result, the latter observes throughput loss (similar to Figure 15). Justitia can effectively provide performance isolation between bandwidth- and throughput-

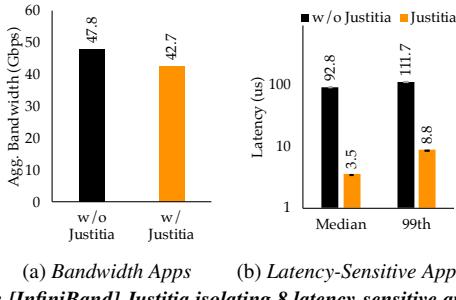


Figure 23: [InfiniBand] Justitia isolating 8 latency-sensitive applications from 8 bandwidth-sensitive ones. Note that 8/9th of the bandwidth share is guaranteed since Justitia counts all latency-sensitive apps as one by default (§4.3.3).

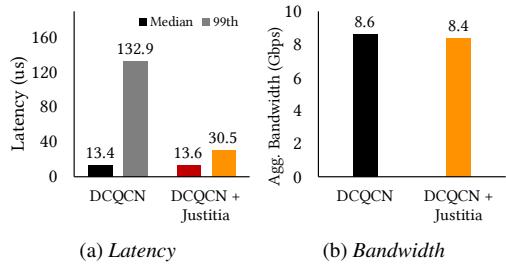


Figure 24: [DCQCN] Incast experiment with 33 senders and a single receiver. 32 senders launch bandwidth-sensitive applications, the other sender launches a latency-sensitive application.

sensitive applications in the shared queue.

6.4 Isolating among More Competitors

We focus on Justitia’s effectiveness in isolating many applications with different requirements and performance characteristics. Specifically, we consider 8 bandwidth-sensitive applications – 2 each with message sizes: 1MB, 10MB, 100MB, and 1GB, and 8 latency-sensitive applications. We measure the latency and bandwidth when all the applications are active in Figure 23. Target_{99} is set to $2\mu\text{s}$ and 20 million samples are collected for latency measurements.

Without Justitia, latency-sensitive applications suffer large performance hits: individually each application had median and 99th percentile latencies of 1.3 and 1.4 μs (Figures 3a and 3b). With bandwidth-sensitive applications, they worsen by $71.4\times$ and $79.8\times$. Justitia improves median and tail latencies of latency-sensitive applications by $26.5\times$ and $12.7\times$ while guaranteeing R_{min} among all the applications.

6.5 Handling Incast with Receiver-Side Updates

So far, we have focused on host-side RNIC contentions where the network fabric is not a bottleneck. We now evaluate how Justitia leverages receiver-side updates to handle receiver-side incast in both RoCEv2 with DCQCN and InfiniBand with its native credit-based flow control. In this experiment, 33 senders are used with the first 32 continuously launch a bandwidth-sensitive application sending 1MB messages to a single receiver. Simultaneously, the last sender launches a latency-sensitive application with messages sent to the same

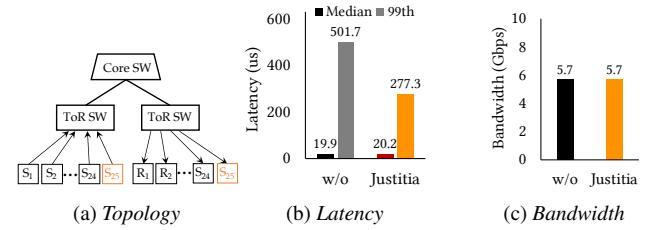


Figure 25: [DCQCN] Justitia’s performance when Inter-ToR links are congested. Justitia achieves the same bandwidth performance because the total amount of bandwidth share on S_{25} is smaller than SafeUtil due to other traffic flowing in the fabric.

receiver. As described in Section 4.5, Justitia daemon at the receiver sends updates to all the senders whenever a sender application starts or exits, resulting in $\frac{1}{32}$ -th of line rate guaranteed at each of the first 32 senders.

Figure 24 plots the results of this experiment, which show that Justitia still reduces tail latency even after the impact of fabric-level congestion on the reference flow latency measurements. Since the monitored latency misses the target, all the bandwidth-sensitive applications send at the minimum guaranteed rate. However, Justitia still achieves high aggregate bandwidth because this is greater than the fair share. This shows that Justitia complements congestion control and further improves the performance of latency-sensitive applications by mitigating receiver-side RNIC congestion.

We have also included a discussion on frequently asked questions regarding reference flows’ impact in large-scale incast scenarios in Appendix E.4.

6.6 Justitia with Unexpected Network Congestion

When there is congestion inside the network, all traffic flowing through the network will experience increased latency, including the packets generated by Justitia as latency signals. Because today’s switches and NICs do not report their individual contributions to end-to-end latency, Justitia cannot tell them apart. However, in practice, this is not a problem because the same response is appropriate in both scenarios.

To evaluate how Justitia performs under such cases, we performed experiments utilizing two interconnected ToR switches on CloudLab [11]. There are servers attached to each ToR switch, and every server has a line rate of 10Gbps. The experiment topology is shown in Figure 25a. In this topology, there is a third core switch that connects to each of the ToR switches with a link with a capacity of 160 Gbps. In this experiment, we enable DCQCN at all the servers and ECN marking at the ToR switches in the cluster.

To create a congested ToR uplink, we launch 24 bandwidth-sensitive applications each issuing 1MB messages from 24 servers (S_1 – S_{24}) under one rack to the other 24 servers (R_1 – R_{24}) under another rack, and none of the servers run Justitia. At the same time, we issue 8 bandwidth-sensitive applications and 1 latency-sensitive application between a pair of servers (S_{25} and R_{25}) that is controlled by Justitia. Figure 25 shows the performance with and without Justitia applied. Even in

the case where fabric congestion is out of Justitia’s control, we see that Justitia can still function correctly, and Justitia still provides additional performance isolation benefits when compared with just using congestion control (DCQCN).

7 Related Work

RDMA Sharing Recently, large-scale RDMA deployment over RoCEv2 have received wide attention [23, 41, 44, 45, 64]. However, the resulting RDMA congestion control algorithms [40, 41, 44, 64] primarily deal with Priority-based Flow Control (PFC) to provide fair sharing between bandwidth-sensitive applications inside the network. In contrast, Justitia focuses on RNIC isolation and complements them (§6.3).

Justitia is complementary to FreeFlow [35] as well. FreeFlow enables *untrusted* containers to securely preserve the performance benefits of RDMA. Because it does not change how verbs are sent to queue pairs, it can still suffer from the performance isolation problems Justitia addresses. Justitia can complement FreeFlow to provide performance isolation by implementing Justitia splitter in FreeFlow’s network library and Justitia daemon in its virtual router.

SR-IOV [55] is a hardware-based I/O virtualization technique that allows multiple VMs to access the same PCIe device on the host machine. Justitia design does not interfere with SR-IOV and will still work on top of it. To provide multi-tenant fairness, Justitia can be modified to distribute credits among VMs via shared memory channel similar to [35].

LITE [62] also addresses resource sharing and isolation issues in RNICs. However, LITE does not perform well in the absence of hardware virtual lanes (Appendix C.4).

PicNIC [38] tries to provide performance isolation at the receiver-side engine congestion in software-based kernel-bypass networks, where it utilizes user-level packet processing instead of offloading packetization to an RNIC. Hence, PicNIC’s CPU-based resource allocation and packet-level shaping cannot be applied to RDMA.

Swift [36] also considers receiver-side enginer congestion in software KBN by using a dedicated enging congestion window in the congestion algorithm. However, both Swift and PicNIC ignores sender-side congestion.

Offloading with SmartNICs Recent research in SmartNICs has focused on providing programmability and efficiency in hardware offloading [6, 19, 33, 39, 42]. However, on-NIC packet orchestration leads to tens of microsecond overhead [19, 57], making performance-related multi-tenancy support still an open problem.

NICA [18] provides isolation for FPGA-based SmartNICs by I/O channal virtualization and time-sharing of the Acceleration Functional Units. Justitia focus on normal RNICs and does not require hardware changes.

Link Sharing Max-min fairness [9, 15, 28, 54] is the well-established solution for link sharing that achieves both sharing incentive and high utilization, but it only considers bandwidth-

sensitive applications. Latency-sensitive applications can rely on some form of prioritization for isolation [3, 25, 63].

Although DRFQ [20] deals with multiple resources, it considers cases where a packet sequentially accessed each resource, both link capacity and latency were significantly different than RDMA, and the end goal is to equalize utilization instead of performance isolation. Furthermore, implementing DRFQ required hardware changes.

Both Titan [58] and Loom [56] improve performance isolation on conventional NICs by programming on-NIC packet schedulers. However, this is not sufficient for RDMA performance isolation because it schedules only the outgoing link. Further, Justitia works on existing RNICs that are opaque and do not have programmable packet schedulers.

TAS [34] accelerates TCP stack by separating the TCP fast-path from OS kernel to handle packet processing and resource enforcement. However, TAS does not solve the type of isolation anomalies Justitia deals with. Justitia’s design idea can be applied to improve isolation for TAS.

Datacenter Network Sharing With the advent of cloud computing, the focus on link sharing has expanded to network sharing between multiple tenants [4, 8, 10, 46, 50, 53]. Almost all of them – except for static allocation – deal with bandwidth isolation and ignore predicted latency on latency-sensitive applications.

Silo [29] deals with datacenter-scale challenges in providing latency and bandwidth guarantees with burst allowances on Ethernet networks. In contrast, we focus on isolation anomalies in multi-resource RNICs between latency-, bandwidth-, and throughput-sensitive applications.

8 Concluding Remarks

We have demonstrated that RDMA’s hardware-based kernel bypass mechanism has resulted in lack of multi-tenancy support, which leads to performance isolation anomalies among bandwidth-, throughput-, and latency-sensitive RDMA applications across InfiniBand, RoCEv2, and iWARP and in 10, 40, 56, and 100 Gbps networks. We presented Justitia, which uses a combination of sender-based resource mediation with receiver-side updates, Split Connection with message-level shaping, and passive machine-level latency monitoring, together with a tail latency target as a single knob to provide network sharing policies for RDMA-enabled networks.

Acknowledgments

Special thanks go to the CloudLab and ConFlux teams for enabling most of the experiments and to RTCL for some early experiments on RoCEv2. We would also like to thank all the anonymous reviewers, our shepherd, Costin Raiciu, and SymbioticLab members for their insightful feedback. This work was supported in part by NSF grants CNS-1845853, CNS-1909067, and CNS-2104243, gifts from VMware and Google, and an equipment gift from Chelsio Communications.

References

- [1] 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 *OSDI*, 2016.
- [2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, , and Michael Wei. Remote regions: a simple abstraction for remote memory. In *ATC*, 2018.
- [3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM*, 2013.
- [4] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O’Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *OSDI*, 2014.
- [5] Apache. Apache crail. <http://crail.incubator.apache.org/..>, 2021.
- [6] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *NSDI*, 2020.
- [7] Infiniband Trade Association. Infiniband architecture specification volume 1. <https://cw.infinibandta.org/document/dl/7859>, 2015.
- [8] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.
- [9] J.C.R. Bennett and H. Zhang. WF²Q: Worst-case fair weighted fair queueing. In *INFOCOM*, 1996.
- [10] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. HUG: Multi-resource fairness for correlated and elastic demands. In *NSDI*, 2016.
- [11] Cloudlab. <http://cloudlab.us/>.
- [12] RL Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.
- [13] RL Cruz. A calculus for network delay, Part II: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991.
- [14] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCaboeter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, , and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *NSDI*, 2018.
- [15] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, 1989.
- [16] Aleksandar Dragojevic, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast remote memory. In *NSDI*, 2014.
- [17] Nick G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, Kadangode K Ramakrishnan, and Jacobus E van der Merwe. A flexible model for resource management in virtual private networks. In *SIGCOMM*, 1999.
- [18] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *USENIX ATC*, 2019.
- [19] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohita, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, , and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *NSDI*, 2018.
- [20] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. 2012.
- [21] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.
- [22] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.

- [23] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *SIGCOMM*, 2016.
- [24] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM*, 2015.
- [25] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *SIGCOMM*, 2012.
- [26] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX ATC*, 2012.
- [27] Intel. HTB Home. <http://luxik.cdi.cz/~devik/qos/htb/>, 2003.
- [28] Jeffrey M Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.
- [29] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *SIGCOMM*, 2015.
- [30] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, 2014.
- [31] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, 2016.
- [32] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter rpcs can be general and fast. In *NSDI*, 2019.
- [33] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *ASPLOS*, 2016.
- [34] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *EuroSys*, 2019.
- [35] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *NSDI*, 2019.
- [36] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G Wassel, Xian Wu, Yaogong Montazeri, Behnam andand Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.
- [37] Gautam Kumar, Srikanth Kandula, Peter Bodik, and Ishai Menache. Virtualizing traffic shapers for practical resource allocation. In *HotCloud*, 2013.
- [38] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Adriaens Jake, Steve Gribble, Nate Foster, and Amin Vahdat. PicNIC: Predictable virtualized nic. In *SIGCOMM*, 2019.
- [39] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang Aditya Akella, Michael M. Swift, and T.V. Lakshman. Uno: Unifying host and smart nic offload for flexible packet processing. In *SoCC*, 2017.
- [40] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M. Swift. RoGUE: RDMA over generic unconverged ethernet. In *SoCC*, 2018.
- [41] Yuliang LI, Harry Hongqiang Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpc: High precision congestion control. In *SIGCOMM*, 2019.
- [42] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *SIGCOMM*, 2019.
- [43] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kokonov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *SOSP*, 2019.
- [44] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM*, 2015.
- [45] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *SIGCOMM*, 2018.
- [46] Jeffrey C Mogul and Lucian Popa. What we talk about when we talk about cloud network performance. *SIGCOMM CCR*, 42(5):44–48, 2012.

- [47] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, 2015.
- [48] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. 2014.
- [49] Marius Poke and Torsten Hoefer. DARE: High-performance state machine replication on rdma networks. In *HPDC*, 2015.
- [50] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.
- [51] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Carlo Contavalli, Amin Vahdat, et al. Carousel: Scalable traffic shaping at end hosts. In *SIGCOMM*, 2017.
- [52] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiyi Zhang. LegoOS: A disseminated, distributed os for hardware resource disaggregation. In *OSDI*, 2018.
- [53] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Sharing the data center network. In *NSDI*, 2011.
- [54] Madhavapeddi Shreedhar and George Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, 1996.
- [55] SR-IOV. Single root i/o virtualization. http://pcisig.com/specifications/iov/single_root/, 2018.
- [56] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient nic packet scheduling. In *NSDI*, 2017.
- [57] Brent Stephens, Aditya Akella, and Michael Swift. Your programmable nic should be a programmable switch. In *HotNets*, 2018.
- [58] Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift. Titan: Fair packet scheduling for commodity multiqueue nics. In *USENIX ATC*, 2017.
- [59] I. Stoica, H. Zhang, and T.S.E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service. In *SIGCOMM*, 1997.
- [60] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of temporary storage in the nodekernel architecture. In *ATC*, 2019.
- [61] Mellanox Technologies. Mellanox PerfTest Package. <https://community.mellanox.com/docs/DOC-2802>, 2017.
- [62] Shin-Yeh Tsai and Yiyi Zhang. LITE kernel rdma support for datacenter applications. In *SOSP*, 2017.
- [63] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.
- [64] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *SIGCOMM*, 2015.

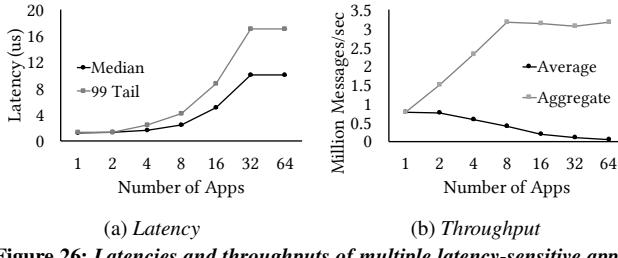


Figure 26: Latencies and throughputs of multiple latency-sensitive applications in InfiniBand. Error bars (almost invisible due to close proximity) represent the applications with the lowest and highest values.

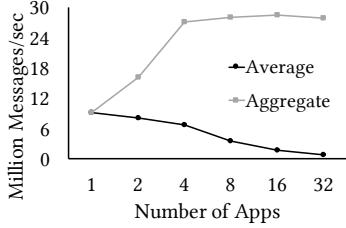


Figure 27: Throughputs of multiple throughput-sensitive applications in InfiniBand. Error bars (almost invisible due to close proximity) represent the applications with the lowest and highest values.

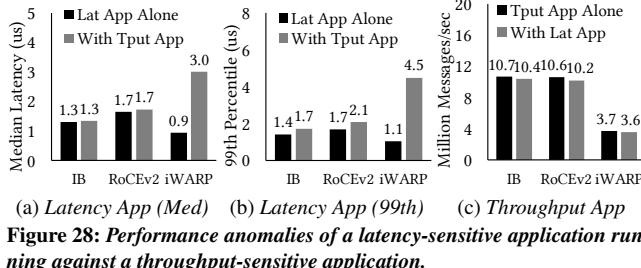


Figure 28: Performance anomalies of a latency-sensitive application running against a throughput-sensitive application.

A Hardware Testbed Summary

Table 1 summarizes the hardware we use for different RDMA protocols in our experiments.

B Characteristics of Latency- and Throughput-Sensitive Applications in the Absence of Bandwidth-Sensitive Ones

Multiple latency-sensitive applications can coexist without affecting each other (Figure 26). Although latencies increase, everyone suffers equally. All applications experience the same throughputs as well.

Similarly, multiple throughput-sensitive applications receive almost equal throughputs when competing with each other, as shown in Figure 27.

Finally, throughput-sensitive applications do not get affected by much when competing with latency-sensitive applications (Figure 28c). Nor do latency-sensitive applications experience noticeable latency degradations in the presence of throughput-sensitive applications except for iWARP (Figure 28a and Figure 28b).

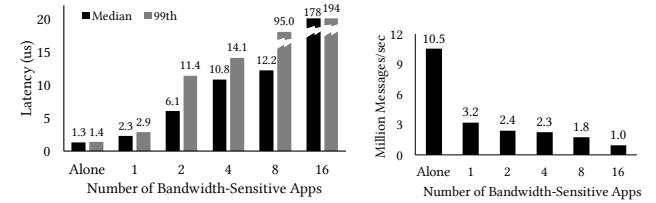


Figure 29: Impact of increasing background bandwidth-sensitive applications (sending IMB messages) in InfiniBand.

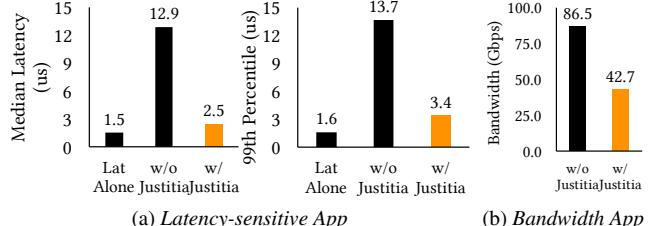


Figure 30: [100 Gbps InfiniBand] Performance isolation of a latency-sensitive application against a bandwidth-sensitive application.

B.1 Adding More Competitors Exacerbates the Anomalies

The lack of protection for the latency-sensitive applications further exacerbates as more bandwidth-sensitive applications (or equivalently more QPs) are created. We increase the number of bandwidth-sensitive applications (each with a single QP) in our experiment to simulate more realistic datacenter applications. Although InfiniBand performs relatively well in the presence of a single background bandwidth-sensitive application (Figure 3), adding one more competitor incurs an additional drop of 2.65× and 3.79× in median and 99th percentile latencies (Figure 29a). With 16 or more bandwidth-sensitive applications, the latency-sensitive application can barely make any progress. We observed a similar trend in other RDMA technologies.

Similarly, a throughput-sensitive application loses 90% of its original throughput with 16 bandwidth-sensitive applications (Figure 29b).

Those anomalies illustrate RNIC’s inability to handle multiple types of applications, which could stem from the limited number of queues inside the RNIC hardware, increasing Head-of-Line blocking of small messages.

C Additional Evaluation Results

C.1 100 Gbps Results With/Without Justitia

Similar to the anomalies observed for 10, 40, and 56 Gbps networks (§3), Figure 30 and Figure 31 show that latency- and throughput-sensitive applications are not isolated from bandwidth-sensitive applications even in 100 Gbps networks. In these experiments, we use 5MB messages since 1MB messages are not large enough to saturate the 100 Gbps link. Justitia can effectively mitigate the challenges by enforcing performance isolation.

Protocol	NIC	Switch	NIC Capacity
InfiniBand	ConnectX-3 Pro	Mellanox SX6036G	56 Gbps
InfiniBand	ConnectX-4	Mellanox SB7770	100 Gbps
RoCEv2	ConnectX-4	Mellanox SX6018F	40 Gbps
RoCEv2 (DCQCN) (§6.3)	ConnectX-4 Lx	Dell S4048-ON	10 Gbps
iWARP	T62100-LP-CR	Mellanox SX6018F	40 Gbps
RoCEv2 (DCQCN) (§6.5 and §6.6)	ConnectX-3 Pro	HP Moonshot-45XGc	10 Gbps

Table 1: Testbed hardware specification.

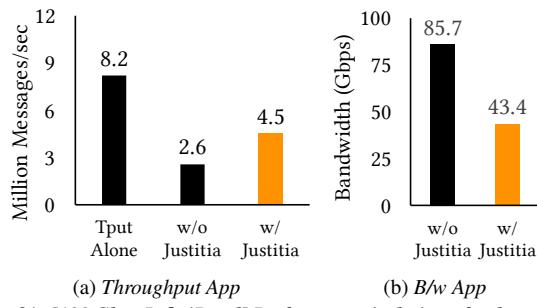


Figure 31: [100 Gbps InfiniBand] Performance isolation of a throughput-sensitive application against a bandwidth-sensitive storage application.

C.2 Real RDMA-based Systems Require Isolation

Besides DARE, highly-optimized RDMA-based RPC systems also suffer from unmanaged RNIC resources. Here we pick two representative systems, FaSST [31] and eRPC [32], to illustrate why they require performance isolation and how Justitia effectively achieves it. To generate background traffic, we implemented a simple RDMA-based blob storage backend across 16 machines. Users read/write data to this storage using a PUT/GET interface via frontend servers. Objects larger than 1MB are divided into 1MB splits and distributed across the backend servers. This generates a stream of 1MB transfers, and the following RDMA-optimized systems have to compete with them in our experimental setup.

FaSST is an RDMA-based RPC system optimized for high message rate. We deploy FaSST in 2 nodes with message size of 32 bytes and a batch size of 8. We use 4 threads to saturate FaSST’s message rate at 9.8 Mrps. In the presence of the storage application, FaSST’s throughput experiences a 74% drop (Figure 32).

eRPC is an even more recent RPC system built on top of RDMA. We deploy eRPC in 2 nodes with message size of 32 bytes. We evaluate eRPC’s latency and throughput using the microbenchmark provided by its authors. For the throughput experiment, we use 2 worker threads with a batch size of 8 on each node because 2 threads are enough to saturate the message rate in our 2-node setting. In the presence of the storage application, eRPC’s throughput drops by 93% (Figure 33b), and its median and tail latencies increase by 67× and 40×, respectively (Figure 33a).

By applying Justitia, FaSST’s throughput improves by 2.5× (Figure 32). Justitia also improves eRPC’s median (tail)

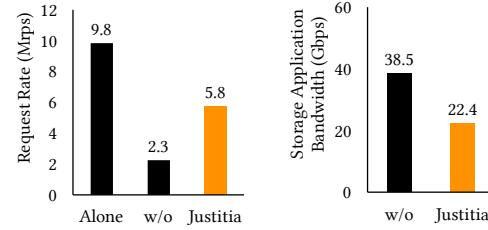


Figure 32: Performance isolation of FaSST running against a bandwidth-sensitive storage application.

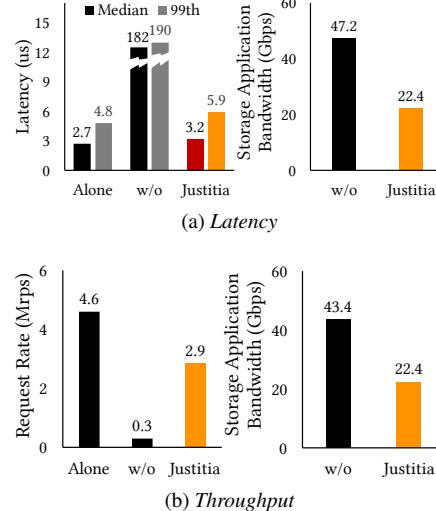


Figure 33: Performance isolation of eRPC running against a bandwidth-sensitive storage application.

latency improves by 56.9× (32.2×) and its throughput by 9.7× (Figure 33). Note that the throughput of the storage applications drops to half of the maximum throughput in both cases because we treat the background application as a whole (and thus with equal weights to all applications, the *SafeUtil* is $\frac{1}{2}$ of the line rate), which is different from how we treat the parallel writes in the case of Apache Crial.

C.3 Handling Remote READs

RDMA READ verbs can compete with WRITEs and SENDs issued from the opposite direction (§4.5) Figure 34 shows that Justitia can isolate latency-sensitive remote READs from local bandwidth-sensitive WRITEs and vice versa.

C.4 Justitia vs. LITE

LITE [62] is a software-based RDMA implementation that adds a local indirection layer for RDMA in the Linux kernel

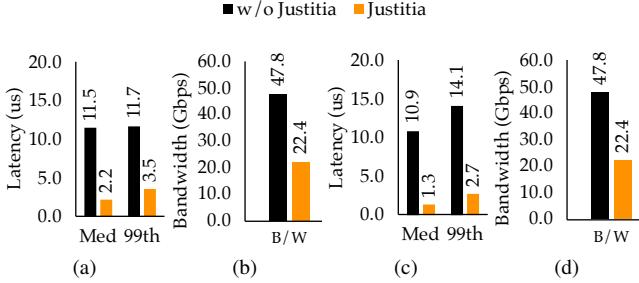


Figure 34: [InfiniBand] (a)–(b) Justitia isolating remote latency-sensitive READs from local bandwidth-sensitive WRITES. (c)–(d) Justitia isolating local latency-sensitive WRITES from remote bandwidth-sensitive READS.

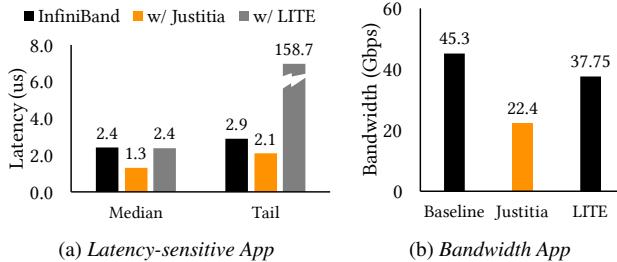


Figure 35: [InfiniBand] Performance isolation of a latency-sensitive flow running against a 1MB background bandwidth-sensitive flow using Justitia and LITE.

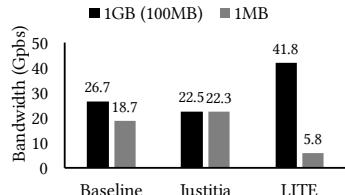


Figure 36: [InfiniBand] Bandwidth allocations of two bandwidth-sensitive applications using Justitia and LITE. LITE uses 100MB messages instead of 1GB due to its own limitation.

to virtualize RDMA and enable resource sharing and performance isolation. It can use hardware virtual lanes and also includes a software-based prioritization scheme.

We found that, in the absence of hardware virtual lanes, LITE does not perform well in isolating latency-sensitive flow from the bandwidth-sensitive one (Figure 35) – 122× worse 99th percentile latency than Justitia. In terms of bandwidth-sensitive applications using different message sizes, LITE performs even worse than native InfiniBand (Figure 36). Justitia outperforms LITE’s software-level prioritization by being cognizant of the tradeoff between performance isolation and high utilization.

D Sensitivity Analysis

Setting Applications Weights To evaluate how assigning different application weights (§4.3.1) affects Justitia’s performance, we launch 4 bandwidth-sensitive applications each sending 1MB message together with a latency-sensitive application, and we vary the weights of the bandwidth-sensitive applications. Figure 37 illustrates the impact of setting different application weights with latency target set to 2 μ s. As

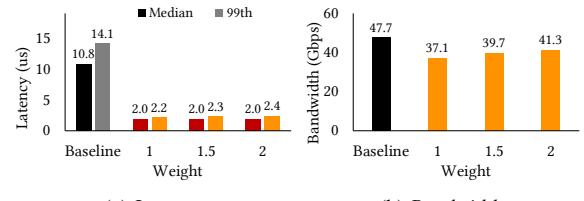


Figure 37: [InfiniBand] Sensitivity analysis of application weights.

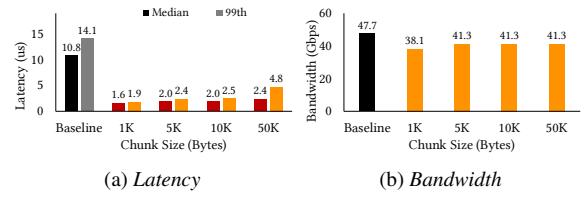


Figure 38: [InfiniBand] Sensitive analysis of chunk sizes.

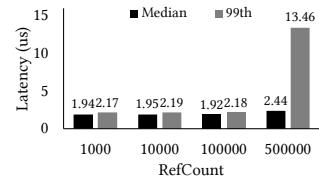


Figure 39: [InfiniBand] Sensitivity analysis of RefCount.

the weight increases, the value of *SafeUtil* increases, and thus more aggregate bandwidth share is obtained for bandwidth-sensitive applications. Higher *SafeUtil* leads to worse latency isolation, but in these experiments the effect of weights on tail latency performance is not huge. In fact, we do not find much latency performance degradation as the weight increases, illustrating the effectiveness of Justitia mitigating head-of-line blocking via its splitting mechanism. The cluster operator can choose weights based on the priority of the applications in the cluster based on the Quality-of-Service inside the cluster (similar to deciding bandwidth resources in a shared environment), or based on how much each application pays to obtain the service. Additionally, if multi-tenant fairness is desired, one can achieve that by modifying how credits are allocated in Justitia on a per-tenant basis. Justitia supports allocating tokens at multiple granularities if needed, which can be per-tenant, per-application, or per group of connections within an application.

Setting Chunk Size When latency-sensitive applications are present, Justitia picks the smallest chunk size that still provides a wide range of bandwidth in case *SafeUtil* is high (Figure 10). Here we evaluate how setting the correct chunk size affects Justitia’s performance. We use the same setting as the sensitivity analysis of application weights and set the weight to be 2. Figure 38 illustrates the experiment results with different chunk sizes. Although a smaller chunk size provides better latency isolation, it is not able to achieve *SafeUtil* and thus waste bandwidth resources. On the other hand, too big of a chunk size does not provide enough latency isolation.

Setting *RefCount* To evaluate how sensitive the value *RefCount* (§4.3.2) is, we design an experiment where initially we launch one latency-sensitive application to compete with a bandwidth-sensitive application. Once the experiment starts, we add three additional bandwidth-sensitive application, with a gap of 1 second between their arrival time. We measure the latency of latency-sensitive application after it completes 10 million messages. Figure 39 plots the results with different *RefCount* values. It turns out that Justitia tracks the tail latency closely as long as *RefCount* is not huge. In Justitia, we set the default value of *RefCount* =10000 to have some memory of latency spikes but not longer enough to impact stable performance.

E Discussion

E.1 Why not simply use hardware priority queues in the RNIC?

Mellanox NICs have priority queues, but as we mention in the paper, the number of queues they support is very limited (e.g., only 2 lossless queues in the RoCE NICs we test out), and we have illustrated such limited number priority queues are insufficient to provide isolation in Figure 23. In addition, the time needed to reconfigure and modify the mapping from applications' QPs to the priority queues is in the order of milliseconds. Last but not the least, it is sometimes also desireable to provide isolation inside a priority level (e.g., bandwidth-sensitive applications and latency-sensitive applications are both assigned with the same QoS level) where hardware priority queues will not be sufficient. Thus, using the priority queues provided by existing hardware does not solve the isolation problem that Justitia faces.

E.2 Why use only 1QP in most of the microbenchmark experiments?

We use a small number of QPs to show that the performance isolation issues can easily occur even with a very small number of active connections. We also test with more number of QPs but the results are placed in Appendix due to limit of space. In fact, adding more QPs exacerbates the performance degradation (Figure 30 in the appendix).

E.3 How does Justitia handle the incast experiment?

Justitia leverages receiver-side updates to make sure the correct minimum rate guarantees are updated correctly at each sender. Due to large latency spike in the case of a network incast, senders will mostly like send via the minimum guaranteed rate (R_{min}) given the latency target will not be met. We discussed receiver-side updates in Section 4.5 and illustrate Justitia complements with existing congestion control and can further help reduce receiver-side engine congestion in Section 6.5.

E.4 Does reference flow and receiver-side updates create additonal congestion in a large scale deployment?

The reference flow sends small messages (10 Bytes every 20 μ s) and only amount to a very small Gbps number (1e6 / 20 * 10 / 1e9 * 8 = 0.004 Gbps), which consumes less than 0.1% of the total link capacity even at nodes with only 10 Gbps link, and thus is not likely to generate any hot spot in the network. When the server broadcasts the receiver-side update, the message is sent using SEND and RECV with a message size of 16 Bytes. With even 1000 client machines this amounts to around 16KB total message size, which is too small to create a potential congestion problem.

In the case of a large-scale latency-sensitive flow incast, if congestion indeed happens, DCQCN will work together with Justitia since it is the major congestion control dealing with fabric congestion. In this scenario, adding more latency-sensitive flows does not prevent Justitia guaranteeing bandwidth share of bandwidth hungry applications.

In the current design of Justitia, the bandwidth-sensitive applications can be rate-limited due to a coexisting latency-sensitive application which is launched at the same host but sends data to a different destination. This is intended behavior to mitigate the anomalies caused by contention at sender-side RNICs, which happens regardless of whether two competing applications are targeting the same receiver. We defer a comprehensive fabric-level solution which involves multiple senders and receives as our future work.

E.5 How to ensure all cooperating SW uses the right protocols and protocol versions?

To deploy Justitia, one only needs to install the Justitia Daemon code and a modified Mellanox driver code on the host machine, and Justitia is compatible with all existing RDMA protocols, including RDMA over Infiniband and RoCE. In datacenter deployments, cluster management tools like Ansible can be used to ensure the appropriate code is deployed at each machine. Additionally, it is straightforward to upgrade Justitia. Because each server in Justitia operates independently, it is not necessary for the same version of Justitia to be deployed across the cluster. Justitia will operate as long as servers are running some version of Justitia.

E.6 How can Justitia be implemented in hardware?

Without having a software layer to split the large RDMA operations before they arrive at the NIC, one probably need to somehow control how the NIC issues PCIe reads. Hardware is often optimized for performance, which in fact is why we are having such isolation issues, so simply decreasing the size of each PCIe reads will definitely affect its maximum throughput performance. To bring Justitia into the hardware design, similar to what we have done in the software layer, the hardware need to recognize when splitting and pacing is needed to provide isolation, and when it should process at

maximum capacity for higher utilization.

E.7 Long-term value of Justitia

As RNICs keep evolving, its performance isolation issues may be mitigated in newer hardware designs. The purpose of this work is to show that there exist such isolation issues in current kernel-bypass networks and illustrate one working approach to mitigate the issue. Design ideas presented in this work can inform hardware designers when developing future RNIC as well as programmable NIC designs.

F Future Research Directions

Interesting short-term improvements of this work include, among others, dynamically determining an application’s performance requirements to handle multi-modal applications, handling idle applications, and extending to more complicated application- and/or tenant-level isolation policies. Long-term future directions include implementing Justitia logic on an RNIC and integrating Justitia with congestion control algorithms.

We highlight these immediate next-steps in the following:

Dynamic Classification (Strategyproof Justitia). Applications may not always correctly or truthfully identify their flow types. To improve Justitia, it is possible to modify the driver to monitor QP usage and automatically identify whether individual connections are bandwidth-, throughput-, or latency-sensitive. This would provide support for multi-modal applications.

Idle Applications. It is straightforward to support idle ap-

plications in Justitia without wasting bandwidth. If a bandwidth hungry app stops sending messages for a long time but does not exit, the driver and daemon can work together to stop token issuing when tokens are not being used and a configurable backlog of tokens has been accumulated.

Justitia at Application and Tenant Levels. Currently, Justitia isolates applications/tenants by treating all flows from the same originator as one logical flow with a single type. However, for an application with flows with different requirements or for a tenant running multiple applications competing with another tenant only running a single application, more complex policies may be desirable. With Justitia, it is straightforward to instead support per-tenant, per-application, or per-flow-group isolation. This is done by allocating tokens at multiple different granularities.

Co-Designing with Congestion Control. Although Justitia effectively complements DCQCN (§6.3) in simple scenarios, DCQCN considers only bandwidth-sensitive flows. A key future work would be a ground-up co-design of Justitia with DCQCN [64] or TIMELY [44] to handle all three traffic types for the entire fabric with sender- and receiver-side contentions (§6.5). While network calculus and service curves [12, 13, 29, 59] dealt with point-to-point bandwidth- and latency-sensitive flows, their straightforward usage can be limited by multi-resource RNICs and throughput-sensitive flows. At the fabric level, exploring a Fastpass-style centralized solution [48] can be another future work.

NetHint: White-Box Networking for Multi-Tenant Data Centers

Jingrong Chen Hong Zhang[†] Wei Zhang Liang Luo[#] Jeffrey Chase Ion Stoica[†] Danyang Zhuo

Duke University [†]UC Berkeley [#]University of Washington

Abstract

A cloud provider today provides its network resources to its tenants as a black box, such that cloud tenants have little knowledge of the underlying network characteristics. Meanwhile, data-intensive applications have increasingly migrated to the cloud, and these applications have both the ability and the incentive to adapt their data transfer schedules based on the cloud network characteristics. We find that the black-box networking abstraction and the adaptiveness of data-intensive applications together create a mismatch, leading to sub-optimal application performance.

This paper explores a white-box approach to resolving this mismatch. We propose NetHint, an interactive mechanism between a cloud tenant and a cloud provider to jointly enhance application performance. With NetHint, the provider provides a *hint* — an *indirect indication* of the underlying network characteristics (e.g., link-layer network topologies for a tenant’s virtual machines, number of co-locating tenants, network bandwidth utilization), and the tenant’s applications then adapt their transfer schedules accordingly. The NetHint design provides abundant network information for cloud tenants to compute their optimal transfer schedules, while introducing little overhead for the cloud provider to collect and expose this information. Evaluation results show that NetHint improves the average performance of allreduce completion time, broadcast completion time, and MapReduce shuffle completion time by $2.7\times$, $1.5\times$, and $1.2\times$, respectively.

1 Introduction

Data-intensive applications (e.g., network functions, data analytics, deep learning) have increasingly moved to the cloud for resource elasticity, performance, security, and ease of management. The performance of the cloud network is critical for these applications’ performance. Cloud providers have thus spent significant effort to optimize various aspects of cloud networks, including network topology [34, 73, 76], congestion control and network stack [3, 33, 42, 44, 69, 77, 92], load balancing [2, 46, 63, 88], bandwidth guarantee [6, 9, 43, 48, 51, 67], debugging [7, 31], fault recovery [53], hardware [8, 27, 52, 58], and virtualization [66].

Today, a cloud provider exposes the network to its tenants as a black box: the cloud tenants have little visibility into their expected network performance (e.g., a constant worst-case bandwidth assurance) or the underlying network characteristics including the link-layer network topology, number of co-locating tenants, and instantaneous available bandwidth.

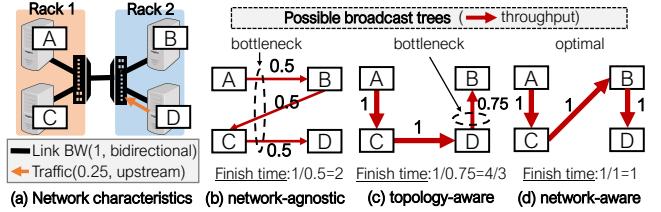


Figure 1: **Applications have the ability and the incentive to adapt their transfer schedules based on network characteristics:** Consider broadcasting a unit-size data object from VM A to VM B, C, and D. (a) shows the network characteristics, all links have bidirectional bandwidth of 1. VM D has upstream background traffic of 0.25. (b) to (d) show possible broadcast trees and their corresponding broadcast finish time. The arrows represent traffic flows and the numbers represent the throughput.

The black-box model has worked well for decades due to its simplicity. However, with the emergence of popular data-intensive applications (e.g., data analytics, distributed deep learning, and distributed reinforcement learning) in the cloud, we observe that such a black-box model is no longer efficient (§2). The crux is that many of these emerging applications have both the *ability* and the *incentive* to adapt their transfer schedules based on the underlying network characteristics, but it is difficult to do so with a black-box network.

Consider broadcast, an important communication primitive in reinforcement learning and ensemble model serving. Figure 1 shows an example that VM A broadcasts to VM B to VM D. Figure 1b shows a possible broadcast tree constructed under the black-box model. Without the underlying network characteristics, the broadcast tree is *network-agnostic*, which introduces link stress on the cross-rack link. Figure 1c shows a broadcast tree based on the topology information (i.e., topology-aware), which improves the broadcast finish time from 2 to $\frac{4}{3}$ time units by minimizing the cross-rack traffic. Figure 1d shows a broadcast tree based on both the topology and bandwidth information (i.e., network-aware). It builds an optimal broadcast tree that avoids the congested upstream link on VM D, further improving the finish time to 1 time unit. The performance gains increase for data center networks that have larger oversubscription ratios or more skewed traffic.

The above example illustrates a fundamental *mismatch* between the black-box nature of existing network abstractions and the ability of a data-intensive application to adapt its traffic. With the black-box model, the cloud tenant is unaware of the network characteristics, and the cloud provider is unaware of the application communication semantics and the transfer schedule. This misses an opportunity for the cloud tenants and

the cloud provider to adapt the data flows to the underlying network topology and conditions to enhance performance and efficiency for these applications. The potential gains are substantial: our benchmark experiment on AWS shows that the allreduce latency for a deep learning experiment varies by up to $2.8 \times$ across different allreduce transfer schedules. One candidate approach is for applications to probe and profile the network and then plan their data flows accordingly [5, 57]. A second option is to report their possible transfer schedules to the provider for the provider to choose. We observe that these alternatives introduce substantial communication latency and system overhead (§2.2).

In this paper, we explore a *white-box* approach to resolve this mismatch. One possibility would be for the cloud provider to expose the physical network topology, the VM locations, along with bandwidth assurances to the application. However, this approach has two major drawbacks. First, exposing VM placement and data center network topology may compromise security for cloud tenants and can raise concerns for the cloud provider (§2). Second, the bandwidth available to a tenant depends on the communication patterns of other tenants, which may be highly dynamic. Predictions that are not timely or not accurate may do more harm than good.

This paper explores an alternative approach. We design and implement NetHint, a mechanism for a cloud tenant and cloud provider to interact to enhance the application performance jointly. The key idea is that the provider provides a *hint* — an *indirect indication* of the bandwidth allocation to a cloud tenant (e.g., a virtual link-layer network topology, number of co-locating tenants, network bandwidth utilization). The tenant applications then adapt their transfer schedules based on the hints, which may change over time. NetHint balances confidentiality and expressiveness: on one hand, the hint avoids exposing the physical network topology or traffic characteristics of other tenants (§9). On the other hand, we show that the hint provides sufficient network information to enable tenants to plan efficient transfer schedules. (§5).

The effectiveness of NetHint relies on addressing three important challenges. First, *what information should the hint contain?* We provide each cloud tenant with a virtual link-layer network topology along with available bandwidth on each link in the virtual topology. This allows applications to adapt their transfer schedules to avoid network congestion.

The second challenge is *how to provide this hint at a low cost*. We design a two-layer aggregation method to collect network statistics on the hosts. We designate a NetHint server in a rack to aggregate network characteristics in the rack. NetHint servers then use all-to-all communication to exchange network characteristics globally. A cloud tenant can thus query its rack-local NetHint server for hints.

The final challenge is *how should applications react to the hint*. We present several use cases for NetHint to optimize communication in a range of popular data-intensive applications including deep learning, MapReduce, and

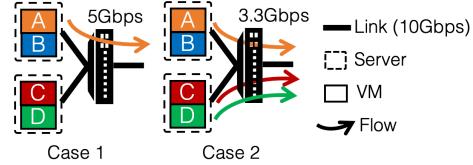


Figure 2: **Examples to illustrate the black-box networking abstraction: tenants cannot predict their network performance.** VM A to D are placed in two servers. All links have 10 Gbps bandwidth. We assume bandwidth is statically partitioned on the end host (each VM can get at most 5 Gbps).

serving ensemble models. The takeaway is that for all these applications, tenants can use the NetHint information via simple scheduling algorithms. Adaptation also has a downside: hints can be stale and adapting transfer schedules based on stale information can hurt performance. We design a policy for applications to adapt flexibly with different hints in different scenarios: applications use temporal bandwidth information when background network conditions are stable and adaptation overhead is low, and otherwise applications fall back to using only the time-invariant topology information (§6).

We evaluate the overheads and the potential performance gain of having NetHint in data centers using a small testbed and large-scale simulations. Our results show that NetHint speeds up the average performance of allreduce completion time in distributed data-parallel deep learning, broadcast completion time in ensemble model serving, and MapReduce shuffle completion time in distributed data analytics by $2.7 \times$, $1.5 \times$, and $1.2 \times$, respectively. Moreover, these benefits are cheap to obtain: NetHint incurs modest CPU, memory, and network bandwidth overheads.

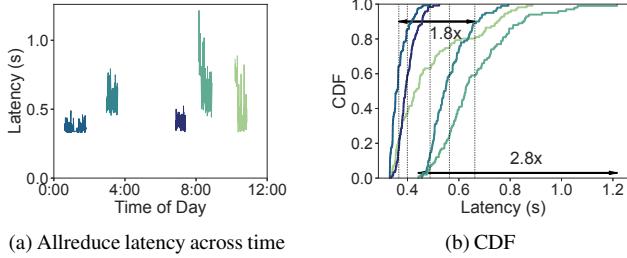
In summary, this paper makes the following contributions:

- We identify a mismatch between the current black-box network abstraction and the communication needs of data-intensive applications.
- We explore a white-box networking approach for multi-tenant data centers.
- We design and implement NetHint, a low-cost system to allow data-intensive applications to adapt their data transfer schedules to enhance performance.

2 Background

2.1 Black-Box Networking Abstraction

Today, the networking abstraction a cloud has is merely a per-VM bandwidth allocation at the end hosts. The abstraction is a *black box*: tenants are unaware of the underlying network characteristics including network topology, number of co-locating tenants, and instantaneous available bandwidth. As a result, the cloud tenants cannot predict their network performance. Figure 2 shows an example. Even with a static allocation of 5 Gbps per VM, VM A cannot predict its network performance because it depends on the traffic demand of other VMs. VM A can get only a bandwidth of 3.33 Gbps when



(a) Allreduce latency across time

Figure 3: Empirical allreduce (256MB) latency of 5 trials. Two trials may have different VM allocations spatially, and each trial contains 100 consecutive runs. (a) shows 5 trials over different times of a day. In (b), each line is the latency CDF of a trial. Each vertical line is the mean latency for a trial. Allreduce latencies vary both across time (up to 1.8x) and across VM allocations (up to 2.8x).

two flows of VM C and D cause congestion inside the network (case 2). Even with work-conserving bandwidth guarantees, a VM’s network performance depends on other VMs.

To quantify this effect, we benchmark allreduce latency on Amazon Web Service (AWS). Allreduce is a collective communication primitive that is commonly used for distributed deep learning. It aggregates a vector (i.e., gradient updates in deep learning) across all worker processes (each running in its own VM). In our experiment, we launch 32 g4dn.2XL (with Linux kernel 5.3) instances in the EC2 US-East-1 region and test ring-allreduce latency with NVIDIA NCCL (version 2.4.8)—the most popular collective communication library for deep learning—for 100 consecutive runs. We repeat the above experiment for 5 trials, and different trials may have different VM placements on the physical topology. Figure 3 shows our findings: ring-allreduce performance on 256MB buffer varies both spatially across different trials and temporally within a trial. Comparing across different trials, the fastest trial has a 1.8× better mean performance than the slowest trial; comparing the 100 runs within a trial, the fastest run is up to 2.8× faster than the slowest run.

2.2 Adaptiveness in Data-Intensive Applications

Besides reinforcement learning and ensemble model serving, which can broadcast model and input data adaptively, as illustrated in Figure 1, we show that many other applications also have both the ability and incentive to adapt their transfer schedules based on the underlying network characteristics.

Many distributed data analytics workloads contain network-intensive shuffle phases between different job stages. For example, the shuffle in MapReduce applications creates an all to all data transfer between the map and reduce stages. The shuffle phase accounts for a large portion of the execution time for many data analytics workloads [16], and numerous studies [4, 15, 16, 39, 84, 87, 90] have demonstrated that optimizing shuffle performance significantly improves application performance. Given network characteristics, distributed data analytics applications can change their transfer schedules (by changing the task placement) to minimize shuffle completion time. Figure 4a

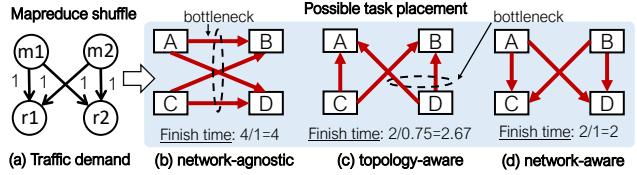


Figure 4: **MapReduce jobs can adapt transfer schedules via task placement.** Assume the same network characteristics as in Figure 1a. (a) shows the traffic demand for a MapReduce shuffle. Each arrow represents a unit traffic. (b) to (d) show possible task placement and the corresponding shuffle finish time.

shows the shuffle traffic for a MapReduce job with two mappers (m1 and m2) and two reducers (r1 and r2). We observe from Figure 4b to Figure 4d that allocating mappers and reducers based on the topology and bandwidth information effectively improves this shuffle completion time from 4 to 2 units. Moreover, emerging task-based distributed systems (e.g., Ray, Dask, Hydro) support applications with dynamic task graphs. Similar to the MapReduce example, we can change the transfer schedule of these applications by choosing different VMs to place a task.

Moreover, many deep learning jobs are network-intensive. This claim is validated by numerous recent studies [14, 35, 40, 71, 86] and observations from production clusters (e.g., Microsoft [30, 41, 82] and ByteDance [65]). In particular, as mentioned in §2.1, deep learning jobs contain an allreduce phase to synchronize gradient updates among workers in each training iteration. As shown in Figure 5, an allreduce phase has multiple candidate topologies. For example, the allreduce traffic can be sent via a ring connecting all the workers with a flexible ordering (Figure 5a and Figure 5b). Or, we can build an allreduce tree to (1) aggregate gradient updates to one of the workers, and (2) send the aggregated gradient updates back in the reverse direction (Figure 5c and Figure 5d). Different allreduce topologies introduce different transfer schedules. Thus, given network characteristics, deep learning jobs can change their transfer schedules by selecting the algorithm and configuration of allreduce.

2.3 Addressing the Mismatch

The black-box nature of the existing networking abstraction and the adaptiveness of data-intensive applications create a mismatch. Data-intensive applications would benefit from more network information from the cloud provider to configure their transfer schedules, but black-box networking hides this information.

Solutions based on the black-box abstraction. There are two approaches to address this mismatch without modifying the existing black-box networking abstraction. One possible approach is to let the cloud provider optimize the communication for tenants as a cloud service. To this end, we first have to develop a general networking API for cloud tenants to express their communication semantics, traffic loads and optimization objectives to the cloud provider. The API design should be similar to the coflow abstraction [16] or the virtual cluster ab-

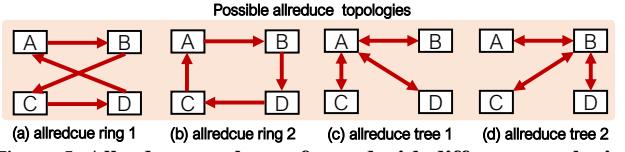


Figure 5: Allreduce can be performed with different topologies. (a) to (d) show 4 possible allreduce topologies to perform allreduce among the 4 VMs (workers).

straction [9], but more general to support a large variety of possible traffic patterns and user-defined objectives. Moreover, a recent measurement study [78] shows that major public clouds exhibit high bandwidth variability at a time granularity of seconds. Thus it is hard, if not impossible, for the cloud provider to perform timely network scheduling for thousands of tenants in a centralized manner, while ensuring network SLAs (e.g., defined via the networking API) for each tenant respectively.

Another potential approach is for cloud tenants to run extensive performance profiling in their allocated VMs [29, 49, 57]. For example, PLink [57] probes the VM pair-wise bandwidth and latency with DPDK and uses K-means clustering to reverse engineer the underlying network topology. This allows it to achieve high allreduce performance by choosing a good allreduce algorithm. Choreo [49] uses 3-step measurements to pinpoint congested links in the data center network to schedule data analytics workloads. Similar approaches were explored decades ago on Internet traffic routing on wide-area overlay networks [5]: picking a high-performance Internet path based on user measurement. Unfortunately, this approach is both costly, as each tenant/user has to profile the network independently, and slow, because the probing phase delays the start of the application. The PLink authors told us that they use 10000 packets to determine bandwidth between a pair of hosts. Choreo generates 3 minutes of probe traffic to infer the network characteristics for 10 VMs.

A white-box network abstraction? Given the deficiencies of the two black-box based approaches, we instead explore a white-box approach: the provider reveals essential information about the network characteristics to the tenant, and the tenants then optimize their transfer schedules accordingly.

One possible way to achieve this objective is for the cloud provider to reveal to a tenant the location of each VM in the physical link-layer network topology, and estimate available bandwidth between each of the VM-pairs. However, this method can raise security and competitive issues. First, exposing VM allocations in the physical network introduces privacy risks for cloud tenants. For example, a malicious user can locate a targeted tenant’s VMs and perform attacks. Second, the exposed VM allocation information can raise competitive concerns for the cloud provider. For instance, this information might be valuable for competitors to learn a cloud provider’s scheduling policies, thus, lowering its competitive advantage. Third, the bandwidth a tenant can acquire depends on the transfer schedules of *all* the tenants, and a single change in transfer schedule of one tenant may trigger a recalculation

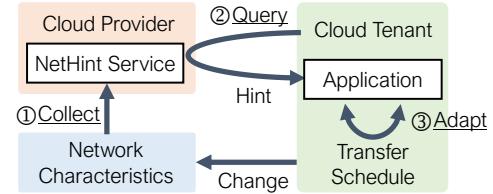


Figure 6: NetHint overview. NetHint service collects network characteristics. Cloud tenants poll hints from NetHint service and adapt their transfer schedules.

for all the tenants. As such, it is computationally expensive for the cloud provider to update the bandwidth shares in real time. Moreover, an application’s bandwidth also depends on *its own* transfer schedule. For example, in Case 2 of Figure 2, if VM A sends one extra flow, the total egress bandwidth of VM A increases to 5 Gbps¹. As a result, without knowing a tenant’s transfer schedule, the cloud provider cannot provide accurate bandwidth estimates to its tenants.

3 NetHint Overview

NetHint is an interactive mechanism between a cloud tenant and a cloud provider to jointly enhance the application performance. The key idea is that the provider provides a *hint* — an *indirect indication* of the underlying network characteristics (e.g., a virtual link-layer topology for a tenant’s VMs, number of co-located tenants, network bandwidth utilization) to a cloud tenant. As illustrated in Figure 6, the provider provides a NetHint service, which periodically (100 ms by default) collects the hint information to capture changes of the underlying network characteristics. A tenant application can query the NetHint service to get the hint information, and then adapt its transfer schedules based on this provided hint. Note that NetHint does not change the fairness mechanism of the underlying network. A tenant can opt in/out any time — whether or not to use NetHint will *not* affect its fair share of the network.

The hint provides a white-box network abstraction which includes additional network information to tenants. As such, users can infer their best transfer schedule without substantial probing latency or communication overhead with the provider. The hint exposes neither the physical network topology nor the location of a tenant’s VMs within it (e.g., which racks). Compared with providing bandwidth information, the hint relieves the provider from the burden of calculating accurate bandwidth allocations. Moreover, compared with calculating bandwidth allocation, it is easier to acquire accurate hint messages (e.g., a virtual link-layer topology for a tenant’s VMs, number of co-located tenants, network bandwidth utilization). As such, the provider is free from the potential risk of providing inaccurate information.

We require NetHint to be: (1) *readily deployable*: all the mechanisms are implementable using commodity hardware; (2) *low cost*: the cloud provider can collect network characteristics with minimal CPU, memory, and bandwidth

¹ Assume per-flow fair sharing in the network.

overheads; (3) *useful*: data-intensive workloads can leverage the hints to achieve high performance. To achieve these goals, NetHint’s design and implementation must address three questions. First, what hints should be provided to the tenants? Second, how should cloud providers collect the hints with low cost? Third, how should applications use the hints to adapt their transfer schedules?

NetHint describes a virtual link-layer topology that connects a tenant’s VMs. In addition, NetHint provides to the tenant recent utilization summaries and counts of co-locating tenant connections on shared network links in the virtual topology. This information allows the tenant to adapt its transfer schedules based on both the topological and temporal hot spots in the network. Further, our design ensures that the only additional information NetHint exposes is aggregated network statistics across all tenants. It is thus difficult for a tenant to acquire information about any individual other tenant. (§4.1)

For the second question, our preferred approach to collecting hints is to measure network traffic in the physical switches using network telemetry, e.g., sketching [54, 55]. However, sketches depend on specific programmable switch features, which are not widely deployed. Instead, our prototype employs a host-driven approach, in which each machine monitors local flows and transmits flow-level statistics to a NetHint measurement plane. One machine in each rack runs a *NetHint server* process to aggregate the rack-level information. These NetHint servers exchange information using periodic all-to-all communication. A cloud tenant connects to the local NetHint server to fetch hints. We show that this approach allows NetHint to provide timely hints to tenants with low CPU and bandwidth overheads (§4.2).

As for the third question, we consider two aspects of adaptation in response to the hints. First, we observe that the adaptation algorithm should take into account the application transfer schedule and semantics to maximize the performance gain. To this end, we consider several use cases for NetHint which cover a range of popular data-intensive applications, including (1) choosing allreduce algorithms in distributed deep learning, (2) constructing broadcast trees for serving ensemble models, and (3) placing tasks in MapReduce frameworks. For each case, we show how applications can adapt their transfer schedules based on the information in the hint. The takeaway is that for all of these examples, tenants can make use of the NetHint information via simple scheduling algorithms (§5).

Second, we explore the drawbacks of adaptation: it introduces extra computational overhead, and may be ineffective or even harmful or unstable if network conditions change too rapidly. We conclude that the adaptation algorithm should use different sets of hints depending on network changing frequency and adaptation overhead. For example, we find that if an application has a non-negligible latency to collect hints and compute the transfer schedules, the bandwidth information may be stale and thus may negatively affect the application performance (detailed in §6). Based on this intuition, we design

Notations & Descriptions

\mathcal{T}	A virtual topology connecting all the tenant’s VMs
l	A virtual link in virtual topology \mathcal{T}
B_e^l	A tenant’s bandwidth share on link l
B_t^l	Total bandwidth on link l
B_r^l	Residual bandwidth on link l
n^l	Number of shared objects on link l

Table 1: Notations and descriptions for NetHint.

a policy for applications to react to hints in a flexible manner: under stable network conditions and low adaptation overheads, applications use both bandwidth and topology information to maximize the performance gain of adaptation. Otherwise, applications use only the stable topology information (§6).

4 Providing NetHint Service

4.1 What Is in the Hint?

NetHint exposes a virtual link-layer topology \mathcal{T} to a cloud tenant. The tenant’s virtual topology abstracts the network as a tree data structure in which the tenant’s VMs are leaf nodes. A link in the tree represents one or more physical links in the data center network, and an interior node may abstract a region of switches and links. The prototype uses a three-layer tree that captures how VMs are distributed among racks in a data center and collapses the network structure above the rack level into a single root node. VMs residing in the same rack are in the same subtree. The virtual topology abstraction does not reveal racks or servers where the tenant has no presence. Following the common observation that congestion losses often occur at the rack level [12, 43, 60, 89], these virtual topologies in the NetHint prototype ignores congestion at any structure above the rack level [23]. It is possible to represent more structure by adding layers to the tree. The tree approximation presumes that the data center network is able to balance its load, so that traffic among children of an abstract node see similar available bandwidth. There is a rich literature on efficient network load balancing for data centers [2, 21, 22, 26, 28, 36, 46, 47, 63, 88], and some of them are readily deployable with commodity hardware.

NetHint allows applications to react to temporal hot spots in the network. For this purpose, NetHint exposes an estimate of utilization on each virtual link l . Recall Case 1 in Figure 2, now assume the orange flow from VM A uses only 2 out of 10 Gbps. If the tenant of VM B knows the network utilization information, it can infer that VM B can send traffic at 8 Gbps. As such, NetHint provides (1) the total bandwidth B_t^l and (2) the residual bandwidth B_r^l on each virtual link l . However, we find that this information alone is insufficient for an application to adapt its transfer schedule, especially when links are congested. For example, even if one link l has already reached 100% utilization, a tenant can still send flows through l and get a fair bandwidth share.

Shared objects and fairness models. In fact, the bandwidth share depends on the fairness model implemented by the cloud provider. Per-flow-fairness and per-VM-pair-fairness

are enforced naturally for RDMA-based networks because modern RDMA NICs can be configured to choose either of them. Per-flow-fairness is ensured for containerized clouds because cloud users cannot modify the kernel TCP stack. For traditional TCP-based and VM-based clouds, many recent studies [18, 37, 62] describe how to enforce per-VM-pair-fairness. With the increasing programmability of modern switches, it now becomes possible to implement other fairness models in the network [74, 83], such as per-tenant fairness.

Consider an application placing 3 connections on a 100 Gbps network link with 7 existing connections from 3 other tenants. We assume each flow can reach 100 Gbps throughput. With per-flow fairness model, the application should get 30 Gbps bandwidth. With per-tenant fairness model, the application should get 25 Gbps bandwidth.

The example indicates that the bandwidth share also depends on the number of *shared objects* on each link l . The definition of shared object depends on the fairness model: it is a flow (VM-pair, tenant) under per-flow (per-VM-pair, per-tenant) fairness, respectively.

To provide bandwidth information, NetHint exposes the number of shared objects n^l on each link l . Taken together, NetHint provides a tuple (n^l, B_t^l, B_r^l) , which includes both the current link utilization and the number of shared objects.

Bandwidth estimation. The information in the virtual topology enables a tenant to estimate its available bandwidth on each virtual link l efficiently. More formally, consider a tenant who plans to place k^l shared objects on link l in its transfer schedule. If link l is an in-network link in virtual topology \mathcal{T} (i.e., not attached to any VM), the bandwidth share the tenant gets can be estimated as:

$$B_e^l = \max\left(\frac{k^l}{n^l + k^l} B_t^l, B_r^l\right) \quad (1)$$

[Equation 1](#) indicates that when the link is under-utilized, the tenant can use up all the residual bandwidth B_r^l , and even if the link is already congested, the tenant can at least achieve its fair share based on the number of shared objects.

If link l is an edge link (i.e., attached to one VM), the bandwidth share is also affected by the underlying sharing approach. More specifically, denote the per-VM bandwidth guarantee provided by the sharing approaches as B_v , we have:

$$B_e^l = \begin{cases} \min(B_v, \max\left(\frac{k^l}{n^l + k^l} B_t^l, B_r^l\right)) & \text{static partitioning} \\ \max\left(\frac{k^l}{n^l + k^l} B_t^l, B_r^l, B_v\right) & \text{work-conserving} \end{cases} \quad (2)$$

Sources and impact of inaccuracy We acknowledge that both [Equation 1](#) and [Equation 2](#) are approximations and can sometimes be inaccurate. First, some shared objects (i.e., tenant, VM-pair, or connection) may have traffic demands less than their fair network share, thus calculating the exact value of B_e^l requires knowing the traffic demand for each shared object. NetHint does not provide per-object information, as doing so introduces security concerns and significant overhead given the huge number of such objects. Second, since a virtual

link corresponds to the aggregation of multiple parallel paths in the physical topology, the estimation may be inaccurate under poor network load balancing across these parallel paths. We note that this is less likely to happen with recently proposed data center network load balancing designs.

Despite these inaccuracies in bandwidth estimation, our results ([§8](#)) show that even the three-level tree approximation is sufficient to adapt the transfer schedules and improve the performance of our target applications. Moreover, evaluation results also show that the benefits degrade gracefully with the quality of the approximations.

Alleviating security and competitive issues. Compared with a naive white-box solution that exposes VM allocation information and physical network topology, NetHint has alleviated the security and the competitive concerns. First, NetHint does not expose the physical location of allocated VMs, so a tenant cannot learn the provider’s VM allocation policy. Second, our network statistics are aggregated over all other tenants, so it is difficult for a tenant to infer from them the network behavior of any other individual tenant. Finally, network topology among a tenant’s VMs is already accessible even in today’s black-box model via user probing approaches, e.g., as presented in PLink [57] and Chereo [49]. NetHint does provide easier access to this information, but we believe this does not increase the security risks. Note that NetHint does not fully eliminate these issues, and we discuss them in [§9](#).

4.2 Timely NetHint with Low Cost

User query overhead The virtual topology is presented as a set of links (each with a Link ID). Each virtual link has its associated B_t . The temporal utilization information for each link includes a tuple of three fields (Link ID, n , B_r). Each field occupies 8 bytes. As such, the amount of data returned by a query is small. Consider a cloud tenant that has rented 100 VMs allocated across 10 racks. As upstream and downstream virtual links are considered separately, the number of virtual links equals twice the sum of the number of VMs and the number of racks the tenant occupies. The amount of query information thus has $(100+10) \times 2 \times 3 \times 8 = 5280$ Bytes.

There is no value or incentive for a tenant to query at a higher frequency than the information update period of NetHint (100 ms by default). Tenant VMs communicate with a NetHint server through TCP connections with rate limits that prevent queries more frequent than once per 50 ms.

Collection overhead We design a two-layer host-driven aggregation approach to collect timely hint information with low cost. Recall that we select one machine in each rack to run a NetHint server process. Each machine collects flow-level network characteristics from its operating system, and sends them to its rack-local NetHint server periodically. The information each machine has to send to the local NetHint server is a virtual link ID plus one (n, B_r) for each virtual machine to ToR link and another (n, B_r) containing only the traffic transmitting

across the rack, for adding its contribution to the ToR uplink's (n, B_r) . Each field is 8 bytes, so the total data size per virtual link is $(1+2\times 2)\times 8=40$ bytes. It is necessary to consider the upstream and downstream bandwidth independently, so each virtual machine or ToR has two associated virtual links. For example, assuming a physical machine has 10 VMs, it sends $40\times 2\times 10=800$ bytes of data to the NetHint server in each period. We set the information update period to 100 ms by default. Thus, the total aggregated information for one NetHint server is two (n, B_r) for every VM-to-ToR virtual link and the ToR uplink in the virtual topology. The NetHint servers then use all-to-all communication to exchange their aggregated information.

Suppose a data center has 1000 racks, and every rack has 20 machines. In each information update period, a local NetHint server gathers 16 KB information (800 bytes \times 20 machines). With a 100 ms update period, the total amount of cross-rack traffic introduced by the all-to-all information exchange is $16\text{ MB}/100\text{ ms} = 1.3\text{ Gbps}$ per rack. Let's assume each rack has outgoing bandwidth of 500 Gbps. Then the bandwidth overhead of NetHint is 0.26%.

Failure detection and recovery NetHint is a best-effort service, and applications should be prepared to function without hints, e.g., if their rack-local NetHint servers become unavailable due to failures such as link failure and server crashing. In this case, applications just revert the transfer schedule to a default one assuming no known network characteristics until a new NetHint server is available in the rack.

5 Adapting Transfer Schedules with NetHint

We find that most data-intensive applications can be categorized into two classes, based on how they can adapt to network characteristics. For each application class, we show that adapting transfer schedules corresponds to an optimization problem. Our goal here is not to present the optimal algorithm to solve the scheduling problems. Rather, our goal is to show that a broad set of distributed applications can benefit from NetHint using simple scheduling algorithms.

5.1 Optimizing Collective Communication

Many data-intensive applications run a high-level collective communication primitive (e.g., broadcast, allreduce) among a set of processes. Any such operation can be accomplished flexibly via a large set of possible *overlay topologies* among all the processes. For example, a broadcast can be performed with different broadcast trees connecting all the receivers, and an allreduce may employ different allreduce topologies (e.g., tree-allreduce or ring-allreduce). For all these communication primitives, the choice of overlay topologies affects only the efficiency (i.e., finish time) but not the correctness. Many popular ML applications belong to this category:

- **Data-parallel deep learning:** each server holds a replica of the model and calculates gradients locally. Servers use allreduce to synchronize gradients in each training iteration.

- **Reinforcement learning:** the trainer process in reinforcement learning repeatedly broadcasts the model (i.e., policy) to a dynamic set of agents.
- **Serving ensemble models:** multiple servers run DNN models simultaneously to predict the label on the same input data, and then use voting to decide the final output. For every input data batch, the front-end server broadcasts it to a set of servers holding different DNNs.

Moreover, as the object of collective communication is usually a vector of numbers, we can partition the object and apply different overlay topologies on each partition. For example, a broadcast can be accomplished via multiple broadcast trees, with each broadcast tree transferring a different (weighted) portion of the broadcast object. Similarly, an allreduce can be performed via a weighed combination of different allreduce topologies. The transfer schedule thus depends on both the choices of overlay topologies and their corresponding weights.

With NetHint, the tenant can estimate the bandwidth B_e^l available on each link l based on [Equation 1](#) and [Equation 2](#). For a transfer schedule s , denote the volume it transfers on each link l as d_s^l . The corresponding latency of the schedule can be estimated as $\max_l(d_s^l/B_e^l)$. Thus, we have:

Problem statement: *Given the virtual topology \mathcal{T} and the estimated bandwidth on each virtual link l , find a transfer schedule that minimizes the latency $\max_l(d_s^l/B_e^l)$.*

To solve the above problem, one major challenge is that the number of candidate transfer schedules can be huge. For example, there can be $O(n^{(n-2)})$ possible broadcast trees to broadcast a message to n processes [79]. One possible solution is to use tree packing algorithms [13, 25, 79]. However, since the goal here is to show the usefulness of NetHint information rather than to find the optimal algorithm, we design simple heuristics to solve the problem. We first sample a random set of overlay topologies (broadcast and allreduce trees) which cross each rack only once. We then use linear programming to find the best weight assignment among these trees, so that the transfer schedule minimizes the latency $\max_l(d_s^l/B_e^l)$.

5.2 Optimizing Task Placement

Many distributed applications execute based on a task graph describing the tasks and their dependencies. The task graph can be static (i.e., task graph is known before the workload runs) [19, 85] or dynamic (i.e., tasks arrive as the workload runs) [61]. Since different tasks may send and receive different amounts of data, the placement of tasks onto VMs determines the transfer schedule among the VMs. Applications in data analytics frameworks and task-based distributed systems therefore can benefit from network-aware task placement:

- **Data analytics frameworks** [32, 85]: data analytics workloads contain network-intensive shuffle phases between different job stages. One shuffle phase creates an all-to-all communication between a set of sender tasks and receiver tasks, so task placement controls the shuffle performance.

Notations & Descriptions	
T_b	Average changing period of the background network condition
T_u	Duration of a transfer schedule being used
T_a	Latency to adapt (collecting information and computing a schedule)
T_s	Staleness of the hint
p	A threshold defined by the ratio between total adapting latency and JCT

Table 2: Important factors related to the impact of staleness.

- **Task-based distributed systems** [38, 61] are increasingly popular in industry. In these applications, the task graph is dynamic and generated at runtime. Tasks launch after fetching input objects from upstream tasks. As such, efficient task placement can minimize the task launch latency reducing the object fetch time.

Problem formulation For both applications, we can formulate the task placement as a classical network embedding problem. Denote the set of tasks as \mathbb{T} and the set of VMs as \mathbb{V} . Compared with the problem statement in §5.1, which selects an efficient data transfer schedule, here we need to find an embedding $\mathcal{E} : \mathbb{T} \rightarrow \mathbb{V}$ given the transfer schedule among all tasks. The algorithm inputs and optimization goals are the same as the problem statement in §5.1, except that the latency is calculated as $\max_l(d_e^l / B_e^l)$. d_e^l is the transfer volume on link l introduced by embedding \mathcal{E} .

We make minor modifications to the greedy heuristics proposed in Hedera [1] to solve the embedding problem. We first sort all tasks in \mathbb{T} based on the amount of data they receive in decreasing order (no need if $|\mathbb{T}| = 1$). We then place tasks one by one following this order. When placing a task to \mathbb{V} , we optimize greedily for the objectives described in the problem statement. Before processing the next task, we update the cross rack traffic and d_e^l based on the placement.

6 Flexible Adaptation for Stale Information

Staleness of NetHint information The staleness of NetHint information during job execution is affected by the following two factors (notations listed in Table 2). First, an application controller can have a non-negligible latency to collect hints and compute the transfer schedules based on the hints, which makes the hints stale when being applied. We denote the adaptation latency as T_a .

Second, applications can adapt to hints periodically. For each adaptation period, the schedule calculated based on the previous hint will be used for the entire duration T_u . Note that for recursive jobs (e.g., model serving), recomputing the schedule for every iteration introduces too much latency. To this end, we fetch hints and recompute the schedule every k iterations, so that the latency to compute transfer schedule is within a portion p (e.g., 10% by default) of the job execution time. Moreover, for jobs that adapt the task placement based on hints (e.g., MapReduce), the adaptation period T_u equals job completion time, as the task placement usually cannot be changed during job execution.

Taken together, the staleness of NetHint information is quantified as $T_s = T_a + T_u$, which is the combination of both above

factors. T_a is the total latency of four steps. The first three steps are to collect hints: sending host network characteristics to NetHint service, NetHint service exchanges rack-level network characteristics, and applications querying the NetHint service. The maximum latency for these three steps combined is 300 ms (100 ms per step due to NetHint frequency), so we use 150 ms as the estimate for the average case latency. The last step is to compute the transfer schedule, and it is application-specific (Figure 8). In our evaluation, a deep learning job of 64 workers requires 10 ms to compute its transfer schedule. We thus set $T_a = 150 + 10 = 160$ ms. We set $T_u = 100$ ms to keep the compute overhead to be less than 10% of the total running time.

Impact of the stale information The impact of stale information depends on the relative relationship between (1) the staleness of the information; and (2) the stability of the underlying network condition. Assume the background network condition changes every T_b time in average. A hint with staleness T_s much less than T_b can still be helpful since the network condition is likely to be similar with the condition T_s time ago. In contrast, a hint with staleness T_s much larger than T_b will be misleading, since the current network condition may be very different from the condition T_s time ago. In this case, adaptation with misleading hints can negatively affect the application performance (Figure 12d).

Flexible adaptation based on application and network condition. There are two takeaways from the above analysis. First, stale information should not be used when it is misleading. Regarding this, one approach is to simply ignore the provided hints and run applications as we run them today. However, as we show in motivating examples (e.g., Figure 1c and Figure 4c), the link-layer network topology alone can be useful for some types of applications to reduce the amount of cross-rack traffic. Compared with the bandwidth information, topology information is more stable and not affected by network dynamics.

Therefore, we propose NetHint-TO, a class of scheduling algorithms that use only the stable topology information from NetHint. For example, with NetHint-TO, we create a ring that crosses each rack only once for ring-allreduce and a chain that crosses each rack only once for tree-broadcast.

The second takeaway is that there is no one-size-fits-all solution. Each application should have two scheduling algorithms, one uses bandwidth information (in §5) and another one uses stable topology information only (NetHint-TO). We design a policy to choose between these two algorithms based on both the application and the network conditions (i.e., T_b , T_u , T_a). More specifically, when $T_s < T_b$, applications use the scheduling algorithm in §5 to calculate the optimal schedule based on both bandwidth and topology information. When $T_s \geq T_b$, applications adopt NetHint-TO to minimize the impact of stale information.

7 Implementation

We implement NetHint using 4600 lines of Rust code. 2300 additional lines of code are in NetHint server to provide NetHint to cloud tenants. The algorithms for applications to adapt transfer schedules (i.e., MapReduce, allreduce, and broadcast) are implemented using 149, 216, and 144 lines of code. We use `lpsolve` [56] for solving linear programs.

To compute the hints in our testbed, we take an endhost-based approach. We hook an eBPF program into the OS kernel. The eBPF program counts the total number of bytes going within the rack and outside the rack. A userspace program polls the counters from the eBPF program every 10 ms and maintains a moving average of the number of existing shared objects (i.e., flows, in a per-flow fairness model). The userspace program sends the number of shared objects and traffic data to the NetHint server every 100 ms. In a deployment environment where SmartNICs is available, we can also program the SmartNICs to implement this logic.

NetHint server binds to a TCP port, where VMs connect to it to fetch hints. NetHint server uses a single thread to respond to NetHint queries. A single thread is enough for our design because queries are not frequent.

For an application to use NetHint, we need to modify the application. For traditional collective communication, the transfer schedule is static and decided before runtime. Recent collective communication designs have shown that transfer schedules can be dynamically decided at runtime [93]. NetHint can help these dynamic collective communication designs to decide on an efficient transfer schedule based on network characteristics. These dynamic collective communication designs can query and adapt transfer schedule every k iterations before issuing data transfer operation. For task placement, the global scheduler of a distributed system (e.g., master in MapReduce [19]) queries the NetHint server and uses both the task information and the NetHint information to decide task placement. For our evaluation purpose, we build a dynamic scheduler for collective communication and a task scheduler for MapReduce tasks according to the descriptions above.

8 Evaluation

8.1 Setup and Workloads

We evaluate NetHint using an on-premise testbed and large-scale simulations. Our setting is that hosts ensure work-conserving bandwidth guarantee for VMs and the network ensures per-flow fairness. We compare NetHint with the scenarios where cloud tenants (1) do not consider network characteristics and (2) probe the network to reverse-engineer the network characteristics and then adapt transfer schedules. For user probing, we assume network information is always correctly reverse engineered. We assume the probing strategy is the following: For a tenant that owns n hosts, user probing runs in $n/2$ rounds, where each round's latency is either the latency to send 10000 packets or 1 second, whichever is smaller, to measure

throughput and latency between $n/2$ pairs of hosts.² Similar to NetHint, user probing adopts the same strategy to periodically update the transfer schedule, but with a lower frequency due to its higher overheads. We calculate user probing's frequency using the same method described in the second paragraph of §6.

We use a mix of two types of background traffic to simulate skewed and long-tailed traffic in data centers [3, 12, 70, 89]. One slow-moving background traffic occupies 0-50% bandwidth of the link capacity on each link in a Zipfian distribution. The slow-moving background traffic occupies 10% bandwidth in total and changes every 10 seconds. The other is a fast-moving background traffic which is on all links and occupies 0-10% bandwidth of the link capacity in a uniform random fashion. The fast changing background traffic changes every 10 ms. We use the following workloads. We run each experiment 5 times and report the average speedup for each job. To quantify the overall speedup, we also measure the arithmetic average of speedups across jobs.

Distributed data-parallel deep learning. We test the allreduce completion time. The job sizes are either 16 or 32 (in terms of number of nodes) with equal probability. For each allreduce job, we set the buffer size to be 100 MB (\approx the size of ResNet-50). We run 100 jobs and assume jobs arrive as a Poisson process. We choose Poisson lambda = 24 seconds, so that the average network utilization approximates to 12%.

Serving an ensemble of ML models. We test the broadcast completion time. We use the same job size distribution described in Hoplite [93]. We run 100 jobs and assume jobs arrive as a Poisson process. We choose Poisson lambda = 8 seconds, so that the average network utilization approximates to 12%.

MapReduce. We test the latency of the data shuffling phase of MapReduce. We use Facebook's MapReduce trace [17], which contains 500 MapReduce jobs and their arrival time. We assume the traffic is divided evenly from a reducer to the mappers.

8.2 NetHint in Testbed Experiments

We build a 6-server testbed. Each server has a 100 Gbps Mellanox ConnectX-5 NIC and two Intel 10-core Xeon Gold 5215 CPUs (2.5 GHz). These machines are connected via an emulated 40 Gbps 2-stage FatTree network using a single 100 Gbps Mellanox SN2100 switch through self-wiring. 3 machines are in one rack, and the rest 3 machines are in the other rack. The oversubscription ratio on our network is 3. Each machine runs 4 VMs where each VM is guaranteed 10 Gbps through fair-queuing on the NICs.

Overheads. We already provide analysis of bandwidth overheads in §4.2. Now the remaining question is how much overhead NetHint incurs in terms of latency and CPU cycles.

²We believe this is a best-case scenario for existing user probing techniques. Plink [57] sends 10000 packets per VM-pair to reverse engineer link-layer topologies. Choreo [49] uses a 3-step strategy to pinpoint congested links and its first step is measure pair-wise bandwidth. It takes 3 minutes to reverse engineer the network conditions for 10 VMs (90 VM-pairs).

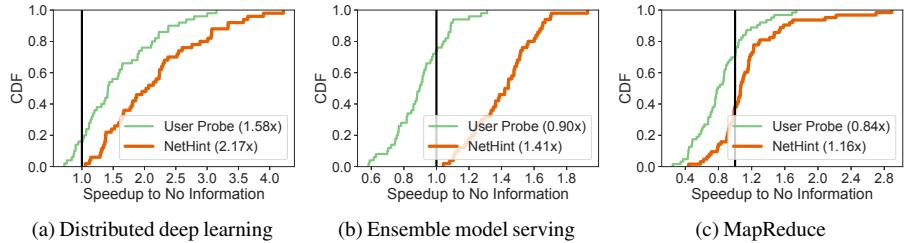


Figure 7: **Testbed results:** NetHint’s speedup on testbed for allreduce in data-parallel distributed training, broadcast in ensemble ML model serving, and mapreduce shuffle compared with user probing and not using network information. Numbers in the legend shows the average of speedups compared with running applications without network information.

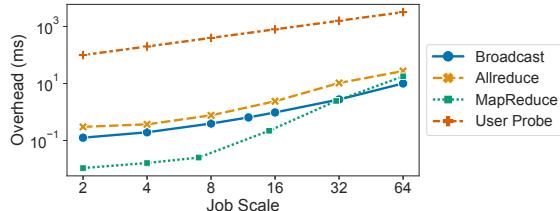


Figure 8: **Testbed results:** Latency to compute transfer schedules.

Collecting statistics from eBPF program is instant, and the polling period for flow statistics is 10 ms.

To measure the overheads in large deployment, we use each CPU core in our testbed to emulate a rack by instantiating a NetHint server per-core. We use pidstat to measure the CPU cycles and memory footprint on NetHint server. Table 3 shows the result. When the number of racks scale up to 240 racks, the CPU time spent on NetHint servers is negligible, i.e., less than 0.66%. The memory footprint on each NetHint server is small (less than 80 MB) and scales with the number of racks mainly due to the increase in the hint size. The latency to collect network information is less than 14 ms.

We implement the algorithms described in §5. We test the computation latency of running each algorithm at different scales (number of workers). Figure 8 presents the results. The latency to make a scheduling decision remains low, ranging from 10 us to 30 ms. Compared with the computation latency, the extra latency introduced by user probing is much higher, ranging from 100 ms to 3 seconds. The round-trip latency to fetch hints takes 100 us because it is rack-local.

Results. NetHint improves application performance. Figure 7 shows the normalized speedup to running applications without network information. Using user probing speeds up the communication by 1.6x for distributed data-parallel deep learning and slows down the communication by 1.1x and 1.2x for serving an ensemble of ML models, and MapReduce shuffle, respectively. NetHint speeds up communication of these workloads by 2.2x, 1.4x, and 1.2x, substantially outperforming user probing. NetHint can outperform user probing because collecting hints is more lightweight than each application individually probing the network characteristics. User probing hurts many ensemble model serving and MapReduce jobs because of the probing overheads. In addition, we notice that a small portion of jobs in Figure 7c are penalized. On our testbed, the job log shows

# Racks	CPU Util. (%)	Memory (MB)	Latency (ms)
6	0.06	4.53	10.60
24	0.14	5.90	10.73
96	0.41	19.28	11.91
240	0.66	78.16	13.73

Table 3: **Testbed results:** The system overhead of a NetHint server in CPU utilization, memory, and information collection latency.

that there are on average 2.8 jobs sharing the rack bandwidth. One job arrival or departure changes the network condition for all the other jobs on the rack. However, the task placement decision cannot be changed during job execution, and thus the initial placement can be imperfect. In contrast, deep learning and model serving workloads in Figure 7 do not severely suffer from this problem, as they can timely modify the transfer schedule for each iteration based on the latest NetHint information.

8.3 NetHint in Simulations

We use simulations to evaluate NetHint in large-scale deployments and in various operating environments. Our simulator is written in 5000 lines of Rust. The simulation is at flow level, and throughput of each flow is the result of solving a max-min fairness formula based on traffic demand. We simulate a CPU cluster and a GPU cluster individually. Both the CPU and GPU clusters have 150 racks. In the GPU cluster network, each rack has 6 machines with 100 Gbps NIC, and each rack has total upstream bandwidth of 200 Gbps. In the CPU cluster network, each rack has 18 machines with 100 Gbps NIC and the total upstream bandwidth is 600 Gbps. The oversubscription ratios are both 3. In the CPU cluster, each machine has 4 VMs. In the GPU cluster, each machine only has 1 VM. All VMs have bandwidth guarantee of 25 Gbps.

Results. Figure 9 shows the NetHint’s speedup of the three workloads in our simulations. In summary, the trend of the simulation results matches what we have observed on the testbed. NetHint speeds up communication by 2.7x, 1.5x, and 1.2x, respectively. On allreduce, the speedup is higher than that on the testbed because the number of hosts involved in a job is larger than that on the testbed, and thus the amount of cross-rack traffic is also larger, giving NetHint more room to optimize transfer schedules.

User probing incurs substantial overheads in both traffic and latency. Figure 10 shows the overheads of using NetHint and

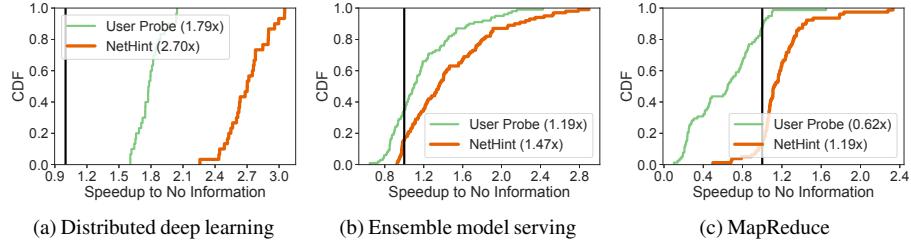


Figure 9: **Simulation results:** Comparing NetHint with dynamic user probe in the default background traffic setting.

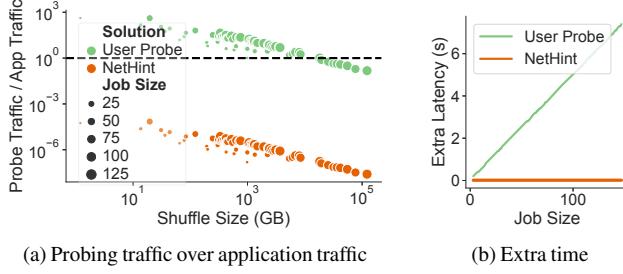


Figure 10: **Simulation results [MapReduce]:** Extra overhead for MapReduce jobs comparing NetHint and user probing.

user probing in MapReduce. The amount of overhead depends on both MapReduce shuffle size and job size. Figure 10a shows the extra traffic introduced by NetHint and user probing over application traffic. NetHint only adds less than 0.1% extra traffic. User probing, in contrast, adds 15% to 420% extra traffic, and 90% of jobs double their traffic. This is because user probing needs to generate probe traffic, and each application has to probe independently. For large shuffle sizes, the probing traffic is less of a concern because it constitutes a smaller fraction of the total traffic. Figure 10b shows the extra latency due to probing and fetching hints for MapReduce jobs of various sizes. NetHint only adds a constant RTT-level extra latency which is negligible. User probing has a large latency overhead, which is linear in job size. This is expected because user probing needs to run for $n/2$ rounds, where n is the job size. There are a set of MapReduce jobs that are penalized substantially by user probing (as shown in Figure 9c). These are MapReduce jobs with large job sizes but with small shuffle sizes.

When should NetHint use topology information only? As we have described in §6, there are two situations we prefer letting NetHint use topology information only: (1) workload granularity is large, and (2) overhead of computing a transfer schedule is non-negligible. To demonstrate these situations, we set the slow-moving background traffic change frequency to every 0.2 seconds. Other environment settings remain the same as those in previous simulations.

To show the case when background traffic changes faster than job completion time, we run 100 broadcast jobs with the model sizes increased to 1 GB. We let NetHint recompute a new broadcast strategy every iteration (but we still guarantee that the computational overhead is under a certain threshold $p = 10\%$). We use NetHint-TO to denote using only topology information when calculating the transfer schedule. We use NetHint-BW to denote using bandwidth information when

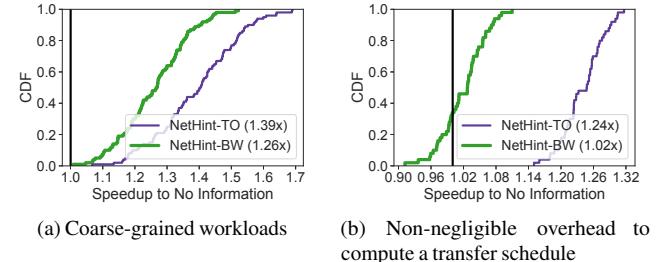


Figure 11: **Simulation results [Model serving]:** Using topology information alone can outperform using bandwidth information.

calculating the transfer schedule. Figure 11a shows that NetHint-TO and NetHint-BW speed up the communication by 1.4x and 1.3x. NetHint-BW is slightly slower than NetHint-TO. Applying a bandwidth-aware algorithm does not bring benefit compared with using topology information only because the background traffic changes even within a single broadcast. Instead, it can slow down the job due to the additional overhead to compute data transfer schedules.

To demonstrate an extreme example for the computational overhead, we run 100 broadcasts of 64 workers with data size set to 12 MB, and we double the bandwidth capacity of ToR switch. Figure 11b shows that NetHint-TO and NetHint-BW speed up by 1.2x and 1.0x compared with no information. NetHint-BW cannot improve because the computation latency using LP is large in contrast to the broadcast latency on such a small data size. It has to adapt its traffic less frequently ($\approx 0.2s$) to ensure the compute overhead is within 10% of the total job completion time. Without being affected by inaccurate hints, NetHint-TO aims to minimize the cross-rack traffic, thus achieving better performance.

Figure 12 shows which adaptation method NetHint choose under different background traffic change periods and oversubscription ratios. The result demonstrates that NetHint chooses the best of NetHint-TO and NetHint-BW for all the three applications we use and also for both oversubscription ratio of 3 and 1.5.

Inaccurate bandwidth estimation. The bandwidth estimations in Equation 1 and Equation 2 is based on approximations, as the accurate estimation requires knowing the traffic demand for each tenant. One question to ask is whether NetHint's design fundamentally relies on the accuracy of bandwidth estimation. To answer this question, we intentionally add noise to the input of NetHint. Having additional noise of $x\%$ means the link utilization provided to NetHint is between $100-x\%$

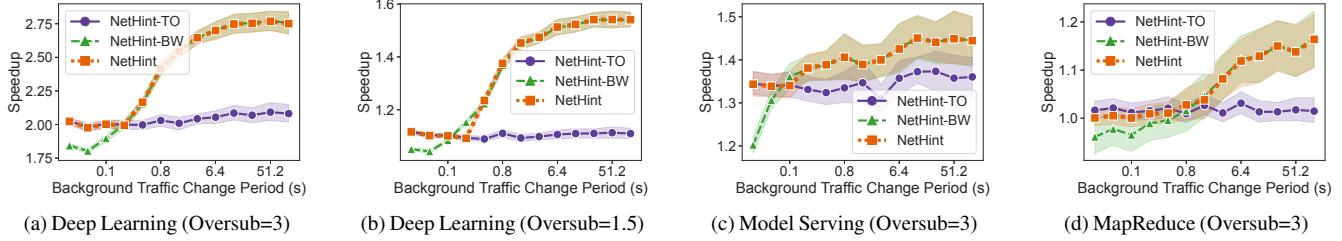


Figure 12: **Simulation results:** Average speedup to background traffic change period under two different topology settings. The shaded area represents 95% confidence interval.

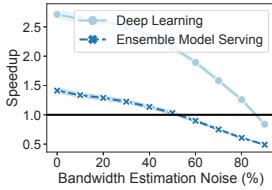


Figure 13: **Simulation results:** Figure 14: **Simulation results:** NetHint’s speedup to not using NetHint’s performance when network information when we varying the number of overlapped add noise to the input of NetHint. jobs.

and $100+x\%$ of the actual utilization. We then evaluate the speedup of allreduce and broadcast jobs. Figure 13 shows the result. NetHint’s speed up degrades gracefully. NetHint can still outperform not using network information when there is up to at most 50% noise.

Performance stability. To evaluate if NetHint’s performance remains stable when the number of NetHint users is large, we increase the number of overlapped jobs. For deep learning, we enlarge the rack size to allow more jobs to share a ToR link and start all the jobs at the beginning. For MapReduce, we scale up the job arrival rate to create more overlapping among jobs. Figure 14 shows that NetHint can constantly achieve performance gain over not using network information.

Sensitivity to network configurations. We evaluate NetHint’s speedup under different network configurations in terms of the number of machines per rack and oversubscription ratios. We vary the number of machines per rack while keeping the oversubscription the same at 3. Figure 15a shows that NetHint can reduce the communication latency consistently for different rack sizes. We then vary the oversubscription ratio. Figure 15b shows that NetHint’s improvement compared with not using network information increases as oversubscription ratio increases. This is because, when oversubscription ratio is high, the cross-rack communication is more likely to become the bottleneck. NetHint can mitigate this bottleneck by reducing the total amount of cross-rack traffic.

Performance gain over perfect user probing. In our evaluation, for n hosts, user probing is performed in $n/2$ rounds. In each round, it measures the bidirectional bandwidth and latency between $n/2$ pairs of hosts in parallel for a certain duration (default to 100 ms). Moreover, we show some evidence that it can be difficult to design better user probing

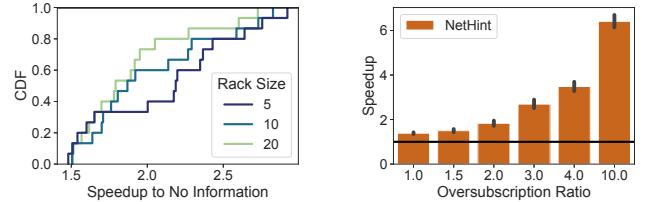
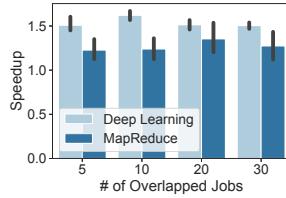


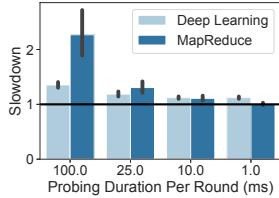
Figure 15: **Simulation results [Distributed deep learning]:** NetHint’s speedup to not using network information when we evaluate under different deployment environments.

technique to achieve similar performance as NetHint. First, we demonstrate how low the user probing duration has to be in order to achieve similar performance as NetHint. For this, we artificially reduce the probing duration while ensuring probing is accurate in simulations. Figure 16a shows the result: even when probing duration is reduced to 1 ms, NetHint still has a small performance advantage over user probing. Second, we show that such a low probing duration (i.e., 1 ms) for accurate bandwidth estimation can be difficult due to data center microbursts. We simulate data center microbursts based on measurement results in Facebook data centers [89] and calculate whether probing for x ms is sufficient to predict the average bandwidth of 100 ms. Figure 16b shows that if we measure for less than 25 ms, there is a 50% probability that the estimation error is above 75%. This is because there are gaps between microbursts, when a busy link is temporarily idle. Probing for such a short amount of time may not detect any traffic.

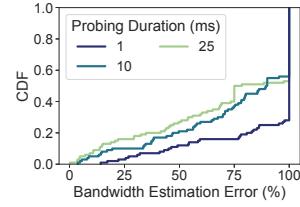
Does NetHint work for other fairness models? The rapid advancement in the programmability in emerging programmable switches makes it possible to implement other types of fairness models in the network [74, 83]. This trend makes it interesting to also understand NetHint’s potential performance gains if we move to other fairness models in the future. We simulate the same allreduce jobs except that we modify our simulator for different fairness models. As shown in Figure 17, the trend of the simulation results matches what we have obtained in a per-flow based fairness setting.

9 Discussion

Herd behaviors. Tenants adapting transfer schedules with provided hints in a distributed way can potentially cause stability issues. For example, given the information of an under-utilized



(a) Sensitivity to probing cost



(b) Probing accuracy

Figure 16: **Simulation results:** The speedup of user probing to NetHint and the relative bandwidth estimation difference under different assumptions of probing durations. The black line in (a) represents NetHint.

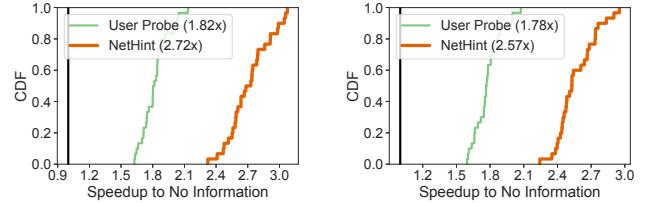
link, many tenants may make identical choices to move traffic to this link, causing congestion. Such herd behavior causes load imbalance and performance oscillation in distributed load balancing problems [2, 59, 88]. We note that herd behavior is a common problem in some specific applications such as distributed load balancers. There are also standard techniques such as adding random jitters, and power of two choices to alleviate herd effect [59]. Whether and how NetHint should help specific applications avoid herd behavior is an interesting future direction. In the workload and setting of our evaluation, NetHint’s speedup does not decrease when we increase the number of overlapped jobs (Figure 14). This infers that the performance of NetHint is not significantly affected by herd behavior.

Other competitive concerns for NetHint. NetHint exposes network utilization information to tenants. Network utilization can be a sensitive information. For example, one can infer whether a cloud has customers and whether a cloud provider does a decent job in network load balancing. NetHint makes it easy for a customer to compare network characteristics at different times. If a customer finds that the achievable bandwidth is reducing via NetHint, there may be a risk that the customer will switch to another cloud provider.

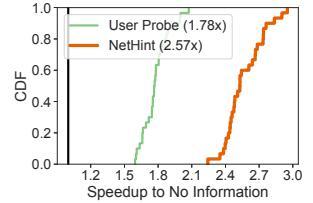
10 Related Work

Sharing network bandwidth. How to share network among many applications or cloud tenants is one of the oldest problems in computer networks. Today, network sharing is opaque to the application or cloud tenants. Within a single tenant, bandwidth sharing is through the fairness property of the underlying congestion control algorithms [24]. Across tenants, a cloud provider usually enforces strong isolation through static bandwidth allocation [68] or work-conserving bandwidth guarantee [9, 10, 50] on the NICs. It is difficult to enable either static bandwidth allocation or work-conserving bandwidth guarantee in the network because commodity switches have limited numbers of hardware queues. NetHint is complementary to these bandwidth sharing design: NetHint does not change any fairness property of the network. NetHint provides guidance for applications to use the network bandwidth better. A non-participating tenant can simply ignore the hint.

Collective communication and task placement based on



(a) Per-tenant fairness



(b) Per-VM-pair fairness

Figure 17: **Simulation results [Distributed deep learning]:** Speedup for other fairness models.

network characteristics. Many related works optimize collective communication [20, 29, 45, 64, 79] or task placement [39, 49, 75, 80, 91] based on topology or bandwidth information. Similar considerations can also be applied inside OS for multi-core machines [11]. Most of these solutions assume the network topology or bandwidth information is already known. As such, NetHint can work in complementary with these solutions by providing them timely network information. Second, these works do not consider a multi-tenant environment. They assume workloads can be controlled by a logically centralized controller, while we assume each tenant’s workload is controlled only by the tenant itself. Because tenants do not know other tenants’ communication patterns, this knowledge needs to be provided either through cloud provider’s support as proposed in this paper or using probing.

User probing. In addition to PLink and Choreo, many past works [5, 72, 81] also propose to measure network characteristics in wide-area networks to choose Internet route. NetHint is different in two aspects: (1) NetHint does not rely on active probe, and thus NetHint has low cost. NetHint simply reads counters directly from NICs or operating systems. (2) NetHint is for distributed applications that can adapt their transfer schedules rather than choosing routes in the network.

11 Conclusion

Today, the networking abstraction a cloud tenant has is a black box. This prevents a tenant’s data-intensive applications from adapting the data transfer schedules to achieve high performance. We design and implement NetHint, a new paradigm for division of work between a cloud provider and its tenants. A cloud provider provides a hint, network characteristics (e.g., a virtual link-layer network topology, number of co-locating tenants, available bandwidth), directly to its tenants. Applications then adapt their transfer schedules based on these hints. We demonstrate the performance gain of NetHint on three use cases of NetHint including allreduce communication in distributed deep learning, broadcast in serving ensemble models, and scheduling tasks in MapReduce frameworks. Our evaluations show that NetHint improves the performance of these workloads by up to 2.7×, 1.5×, and 1.2×, respectively. Our source code is available at <https://github.com/crazyboycjr/nethint>.

Acknowledgement

We thank our shepherd John Wilkes and the anonymous NSDI reviewers for their insightful feedback. We thank Alvin R. Lebeck and Xiaowei Yang for their feedback on earlier versions of the paper. Our work is partially supported by an Amazon Research Award, a Meta Research Award, and an IBM Academic Award.

References

- [1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.
- [2] Mohammad Alizadeh, Tom Edsall, Sarang Dhamapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.
- [3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.
- [4] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*, 2010.
- [5] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *SOSP*, 2001.
- [6] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O’Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI*, 2014.
- [7] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically Finding the Cause of Packet Drops. In *NSDI*, 2018.
- [8] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A Flat Datacenter Network with Nanosecond Optical Switching. In *SIGCOMM*, 2020.
- [9] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.
- [10] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O’Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *NSDI*, 2013.
- [11] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.
- [12] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.
- [13] Chandra Chekuri and Kent Quanrud. Near-linear time approximation schemes for some implicit fractional packing problems. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 801–820. SIAM, 2017.
- [14] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI*, 2014.
- [15] Mosharaf Chowdhury and Ion Stoica. Efficient Coflow Scheduling Without Prior Knowledge. In *SIGCOMM*, 2015.
- [16] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient Coflow Scheduling with Varys. In *SIGCOMM*, 2014.
- [17] Coflow-Benchmark. <https://github.com/coflow/coflow-benchmark>, 2020.
- [18] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. Virtualized Congestion Control. In *SIGCOMM*, 2016.
- [19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
- [20] Mathijs Den Burger, Thilo Kielmann, and Henri E Bal. Balanced Multicasting: High-Throughput Communication for Grid Applications. In *SC*, 2005.
- [21] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Komella. On the Impact of Packet Spraying in Data Center Networks. In *INFOCOM*, 2013.
- [22] Vanini Erico, Pan Rong, Alizadeh Mohammad, Taheri Parvin, and Edsall Tom. Let it Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *NSDI*, 2017.

- [23] Introducing data center fabric, the next-generation Facebook data center network. <https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/>, 2020.
- [24] S. Ben Fred, T. Bonald, A. Proutiere, G. Régnie, and J. W. Roberts. Statistical Bandwidth Sharing: A Study of Congestion at Flow Level. In *SIGCOMM*, 2001.
- [25] Harold N Gabow and KS Manu. Packing Algorithms for Arborescences (And Spanning Trees) In Capacitated Graphs. *Mathematical Programming*, 82(1):83–109, 1998.
- [26] Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, and Mohammad Alizadeh. JUGGLER: A Practical Reordering Resilient Network Stack for Datacenters. In *EuroSys*, 2016.
- [27] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *SIGCOMM*, 2016.
- [28] Soudeh Ghorbani, Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Micro Load Balancing in Data Centers with DRILL. In *HotNets*, 2015.
- [29] Y. Gong, B. He, and J. Zhong. Network Performance Aware MPI Collective Communication Operations in the Cloud. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):3079–3089, 2015.
- [30] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, 2019.
- [31] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM*, 2015.
- [32] Apache Hadoop. <https://hadoop.apache.org/>, 2020.
- [33] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *SIGCOMM*, 2017.
- [34] Vipul Harsh, Sangeetha Abdu Jyothi, and P. Brighten Godfrey. Spineless Data Centers. In *HotNets*, 2020.
- [35] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *MLSys*, 2019.
- [36] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM*, 2015.
- [37] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *SIGCOMM*, 2016.
- [38] Hydro. <https://github.com/hydro-project>, 2020.
- [39] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *SIGCOMM*, 2015.
- [40] Anand Jayaraman, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *MLSys*, 2019.
- [41] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *ATC*, 2019.
- [42] Eun Young Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.
- [43] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.
- [44] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *NSDI*, 2019.
- [45] Nicholas T Karonis, Bronis R De Supinski, Ian Foster, William Gropp, Ewing Lusk, and John Bresnahan. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *IPDPS*, 2000.
- [46] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-Aware Load Balancing at the Virtual Edge. In *CoNEXT*, 2017.

- [47] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *SOSR*, 2016.
- [48] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. PicNIC: Predictable Virtualized NIC. In *SIGCOMM*, 2019.
- [49] Katrina LaCurts, Shuo Deng, Ameesh Goyal, and Hari Balakrishnan. Choreo: Network-Aware Task Placement for Cloud Applications. In *IMC*, 2013.
- [50] Vinh The Lam, Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. Netshare and Stochastic Netshare: Predictable Bandwidth Allocation for Data Centers. *SIGCOMM Comput. Commun. Rev.*, 2012.
- [51] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-Driven Bandwidth Guarantees in Datacenters. In *SIGCOMM*, 2014.
- [52] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit Switching Under the Radar with REACToR. In *NSDI*, 2014.
- [53] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A Fault-Tolerant Engineered Network. In *NSDI*, 2013.
- [54] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *SIGCOMM*, 2019.
- [55] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *SIGCOMM*, 2016.
- [56] Lpsolve. http://web.mit.edu/lpsolve_v5520/doc/index.htm, 2020.
- [57] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. PLink: Discovering and Exploiting Locality for Accelerated Distributed Training on the Public Cloud. In *MLSys*, 2020.
- [58] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. RotorNet: A Scalable, Low-Complexity, Optical Datacenter Network. In *SIGCOMM*, 2017.
- [59] Michael Mitzenmacher. How Useful Is Old Information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–20, 2000.
- [60] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *SIGCOMM*, 2018.
- [61] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*, 2018.
- [62] Zhixiong Niu, Hong Xu, Peng Cheng, Qiang Su, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. NetKernel: Making Network Stack Part of the Virtualized Infrastructure. In *USENIX ATC*, 2020.
- [63] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless Datacenter Load-balancing with Beamer. In *NSDI*, 2018.
- [64] Pitch Patarasuk and Xin Yuan. Bandwidth Efficient All-reduce Operation on Tree Topologies. In *IPDPS*, 2007.
- [65] Y Peng, Y Zhu, Y Chen, Y Bao, B Yi, C Lan, C Wu, and C Guo. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *SOSP*, 2019.
- [66] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.
- [67] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *SIGCOMM*, 2013.
- [68] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *SIGCOMM*, 2007.
- [69] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, 2011.
- [70] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network’s (Datacenter) Network. In *SIGCOMM*, 2015.
- [71] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter

- Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. Technical report, KAUST, Feb 2019. <http://hdl.handle.net/10754/631179>.
- [72] S. Savage, T. Anderson, Amit Aggarwal, David Becker, N. Cardwell, A. Collins, Eric Hoffman, John Snell, Amin Vahdat, G. Voelker, and J. Zahorjan. Detour: Informed Internet Routing and Transport. *IEEE Micro*, 19:50–59, 1999.
- [73] Brandon Schlinker, Radhika Niranjan Mysore, Sean Smith, Jeffrey C. Mogul, Amin Vahdat, Minlan Yu, Ethan Katz-Bassett, and Michael Rubin. Condor: Better Topologies Through Declarative Design. In *SIGCOMM*, 2015.
- [74] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *NSDI*, 2018.
- [75] Haiying Shen, Ankur Sarker, Lei Yu, and Feng Deng. Probabilistic Network-Aware Task Placement for MapReduce Scheduling. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 241–250. IEEE, 2016.
- [76] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, 2012.
- [77] Shin-Yeh Tsai and Yiyi Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *SOSP*, 2017.
- [78] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is Big Data Performance Reproducible in Modern Cloud Networks? In *NSDI*, 2020.
- [79] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and Generic Collectives for Distributed ML. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *MLSys*, 2020.
- [80] R. Wang, J. A. Wickboldt, R. P. Esteves, L. Shi, B. Jennings, and L. Z. Granville. Using Empirical Estimates of Effective Bandwidth in Network-Aware Placement of Virtual Machines in Datacenters. *IEEE Transactions on Network and Service Management*, 13(2):267–280, 2016.
- [81] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Gener Comput Syst*, 15(5–6):757–768, October 1999.
- [82] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI*, 2018.
- [83] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. Twenty Years After: Hierarchical Core-Stateless Fair Queueing. In *NSDI*, 2021.
- [84] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.
- [85] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 2016.
- [86] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *ATC*, 2017.
- [87] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark. In *SIGCOMM*, 2016.
- [88] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient Datacenter Load Balancing in the Wild. In *SIGCOMM*, 2017.
- [89] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-Resolution Measurement of Data Center Microbursts. In *IMC*, 2017.
- [90] Yangming Zhao, Kai Chen, Wei Bai, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. RAPIER: Integrating Routing and Scheduling for Coflow-aware Data Center Networks. In *INFOCOM*, 2015.
- [91] Yangming Zhao, Chen Tian, Jingyuan Fan, Tong Guan, and Chunming Qiao. RPC: Joint Online Reducer Placement and Coflow Bandwidth Scheduling for Clusters. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018.
- [92] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, 2015.
- [93] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems. In *SIGCOMM*, 2021.

Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing

Chaoliang Zeng^{1*} Layong Luo² Teng Zhang² Zilong Wang^{1*} Luyang Li^{3*} Wenchen Han^{4*}
Nan Chen² Lebing Wan² Lichao Liu² Zhipeng Ding² Xiongfei Geng² Tao Feng²
Feng Ning² Kai Chen¹ Chuanxiong Guo²

¹Hong Kong University of Science and Technology ²ByteDance ³ICT/CAS ⁴Peking University

Abstract

Stateful layer-4 load balancers (LB) are deployed at datacenter boundaries to distribute Internet traffic to backend real servers. To steer terabits per second traffic, traditional software LBs scale out with many expensive servers. Recent switch-accelerated LBs scale up efficiently, but fail to offload a massive number of concurrent flows into limited on-chip SRAMs.

This paper presents Tiara, a hardware architecture for stateful layer-4 LBs that aims to support a high traffic rate (> 1 Tbps), a large number of concurrent flows ($> 10M$), and many new connections per second ($> 1M$), without any assumption on traffic patterns. The three-tier architecture of Tiara makes the best use of heterogeneous hardware for stateful LBs, including a programmable switch and FPGAs for the fast path and x86 servers for the slow path. The core idea of Tiara is to divide the LB fast path into a memory-intensive task (*real server selection*) and a throughput-intensive task (*packet encap/decap*), and map them into the most suitable hardware, respectively (i.e., map *real server selection* into FPGA with large high-bandwidth memory (HBM) and *packet encap/decap* into a high-throughput programmable switch). We have implemented a fully functional Tiara prototype, and experiments show that Tiara can achieve extremely high performance (1.6 Tbps throughput, 80M concurrent flows, 1.8M new connections per second, and less than 4 us latency in the fast path) in a holistic server equipped with 8 FPGA cards, with high cost, energy, and space efficiency.

1 Introduction

Large service providers deploy various services inside their geo-distributed datacenters of different scales. At the boundary of these datacenters, stateful layer-4 load balancers (LB), a.k.a., multiplexers (Mux), are deployed to distribute user requests from the Internet to many real servers inside datacenters while preserving connection consistency. Driven by

exponentially increased content delivery and cloud computing demands, a typical LB in large service providers usually has to process terabits per second of Internet traffic, with tens of millions of concurrent flows [25, 31] and millions of new connections per second (CPS) [12].

To support such high performance, vendor-proprietary hardware LBs (e.g., F5 [9]) were deployed in the early days of some datacenters. However, they lacked agility, which is highly desirable in modern hyper-scale datacenters. In recent years, the move from vendor-proprietary hardware to in-house software LBs, or software Muxes (SMux), e.g., Ananta [36] and Maglev [21], was mainly driven by requirements like manageability, reliability, and agility, but sacrificed efficiency (i.e., cost, energy, and space efficiency). For example, Ananta [36] achieves 10 Gbps per instance, and supporting up to terabits per second throughput requires scale-out with a large number of servers. Deploying so many servers for just LB is not only costly but also challenging at energy- or space-limited boundaries of massive small/medium-scale datacenters (e.g., 10s-100s of servers in PoPs [15] or edge [40]). Moreover, software LBs usually suffer from high latency and jitter, sometimes comparable to Internet access latency (in the order of milliseconds [24]) when CPU load is high. Such latency churn will adversely impact users' network experience.

To improve the efficiency of software LBs without sacrificing agility, there is an emerging trend to accelerate software LBs with in-house software and hardware co-design. Recent work [16, 23, 24, 31] leverages programmable switches to accelerate LBs. Nevertheless, programmable switches have inherent scalability issues (§2.3). *On the data plane*, a modern switch cannot store a large number of concurrent flows due to its small memory size (typically 50-100 MB on-chip SRAMs); *on the control plane*, the switch fails to support a large CPS given its slow entry insertion speed (~ 100 Kps).

Existing switch-accelerated LBs do not address both challenges simultaneously. For example, Silkroad [31] stores a small hash of a connection instead of the 5-tuple to compress the connection table. However, its scalability is still bounded by the switch's small memory size, and it may suffer from

* This work is done while Chaoliang Zeng, Zilong Wang, Luyang Li, and Wenchen Han are interns in ByteDance.

throughput reduction due to switch pipeline folding. Moreover, Silkroad does not address the scalability problem on the control plane. Cheetah [16] provides a fast entry insertion mechanism by storing an index in the packet header but requires modifications on services’ client sides. Thus, applying such a mechanism is difficult, if not impossible, in the datacenter with thousands of services [19, 36]. Furthermore, it does not address the scalability issue on the data plane.

One plausible approach to address the above problems is to leverage traffic locality in hardware offloading. If the traffic pattern follows a long-tail distribution (i.e., a small number of flows carry the majority of the traffic), only a few elephant flows need to be offloaded and stored in the switch, thus lowering the requirements of both the hardware memory size and entry insertion speed. However, we observe from production datacenters that the traffic patterns at datacenter boundaries do not necessarily follow a long-tail distribution. Instead, the mix of VIP traffic for multiple services is highly dynamic and unpredictable, detailed in §2.2.

Based on the above analysis and observation, we ask: *can we design a scalable and efficient stateful LB without any assumption on traffic patterns?* Specifically, the design should be:

- **scalable** on both data plane (store > 10M concurrent flows) and control plane (support > 1M CPS);
- **efficient** in terms of high cost, energy, and space efficiency; and
- **generic** without any assumption on traffic patterns.

To this end, we move one step further beyond the existing *switch-server* architecture [23, 24] by exploring more flexible hardware, i.e., FPGA. FPGA is a high-performance and programmable device becoming an important building block in the datacenter infrastructure [22, 28, 29, 42]. The modern FPGA equipped with gigabytes of high-bandwidth memory (HBM) is well-suited to improve LB scalability, as HBM can store a large number of concurrent flows with high lookup and insertion rate.

In this paper, we present Tiara, a three-tier hardware acceleration architecture composed of a programmable switch, FPGAs, and commodity servers, for a high-performance stateful LB with scalability and efficiency. The core idea behind Tiara is that we map different LB tasks into different devices by matching task requirements with device capabilities (§3.1). Specifically, Tiara divides the LB fast path into *real server selection*, a memory-intensive task with both large capacity and high bandwidth requirements, and *packet encap/decap*, a throughput-intensive task. Then, Tiara maps these two tasks into FPGAs with large HBM and a programmable switch with high packet processing throughput, respectively. Similar to other hardware-accelerated systems [23, 24, 37], Tiara leverages commodity servers as the slow path to handle the unprocessed traffic from the fast path.

To support high CPS without compromising line-rate

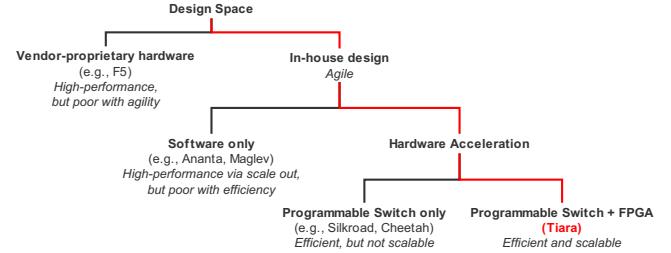


Figure 1: Design space for stateful LB architectures.

packet processing in a heterogeneous system, we optimize several key design components in Tiara (§3.3). First, for both fast lookup and insertion, Tiara adopts fixed-length hash chaining, which leverages the parallel processing capability in both FPGAs and multi-core servers. Second, we design a lock-free offloading approach to support issuing millions of entry operations per second from a server to an FPGA. Third, Tiara employs a lightweight aging mechanism to recycle outdated entries, where FPGAs periodically report connection active-ness via a dedicated accessing bitmap, preventing interference with the data plane.

We have implemented a fully functional Tiara prototype based on a Barefoot Tofino switch, a Xilinx FPGA-based SmartNIC card, and a commodity server. We modified a production-level SMux for the slow path and the control plane (§4). The key results from our experiments (§5) show that our prototype can support 10M concurrent flows and 1.8M CPS, 9× better than Silkroad [31], at 200 Gbps with less than 4 us average latency and small jitter in the fast path. In a holistic server with 8 FPGA cards, Tiara can provide superiority in throughput (up to 1.6 Tbps) and flow capacity (up to 80M concurrent flows). Meanwhile, Tiara achieves 17.4×, 12.8×, and 16.8× higher cost, energy, and space efficiency, respectively, compared to SMux.

As a summary, Figure 1 shows the design space for stateful LB architectures and the unique position of Tiara. Tiara is more agile than traditional vendor-proprietary hardware, faster and more cost-, energy-, and space-efficient than software LBs, and more scalable than switch-accelerated solutions. Specifically, Tiara makes the following contributions:

- We propose a three-tier architecture that matches key LB tasks to the most suitable hardware: programmable switch for packet encap/decap, FPGA with HBM for connection management, and x86 CPU for SMux.
- We design and optimize key LB components, including an efficient hash table structure for fast lookup and efficient insertion, a lock-free offloading approach to improve connection offloading speed, and a lightweight aging mechanism with little overhead and minimal interference on the data plane.
- We implement the Tiara prototype and conduct testbed experiments to show its performance superiority.

2 Background

2.1 Layer-4 Load Balancing

Layer-4 LB can be classified into the stateful LB, which stores the connection-to-real server (RS) mapping as a connection table (CT), and the stateless LB, which does not maintain any per-connection state. Most of the industry LBs are stateful [3, 5, 8, 21, 36] because stateful LBs can easily ensure *per connection consistency (PCC)* [16, 31], which means all packets of a connection should be delivered to the same RS to avoid breaking the connection. In this paper, we focus on the stateful LB, which usually contains the following two parts.

Real server selection: The LB selects an RS for each incoming packet by identifying its connection via the 5-tuple in the packet header. The LB selects RS in two ways. For the first packet of a connection, the LB selects an RS based on a pre-defined algorithm, e.g., hash, round-robin, or least-loaded, and creates a connection entry in the CT to record this selection. The LB selects the same RS for the other packets of this connection by looking up the CT. This mechanism ensures PCC. An RS can be specified by a tuple of $\{\text{RS_IP}, \text{RS_Port}\}$ based on backend service implementations.

Packet encaps/decap: After an RS is selected for an inbound packet, the LB encapsulates the packet with RS_IP and RS_Port. The encapsulation process may include multiple steps in practice. Given a tuple of $\{\text{RS_IP}, \text{RS_Port}\}$, the LB enforces Port NAT (virtual Port (VPort) \rightarrow RS_Port), IP NAT (virtual IP (VIP) \rightarrow RS_IP), and packet encapsulation with VxLAN. Unlike inbound traffic processing involving both RS selection and packet encapsulation, outbound traffic processing only needs packet decapsulation.

2.2 Nature of Internet traffic at the Datacenter Boundary

Large service providers usually deploy many Internet services in a datacenter, and the Internet traffic at the datacenter boundary is a mix of multiple services' traffic, with the following properties.

The flow distribution of individual services varies. The distribution of service traffic depends heavily on the service's client- and server-side implementations. For example, certain service clients may split an elephant flow into multiple smaller ones to reduce the cost of TCP disconnection over unstable Internet, leading to a uniform distribution. In contrast, other service clients may use short connections for synchronization and long connections for massive data transmission, leading to a long-tail distribution. To show this fact, we analyze flow distributions for three different services, as shown in Figure 2. These three services have various flow distributions: service C shows a uniform distribution (where top 10% flows carry 19.6% traffic), while service A and B exhibit traffic locality

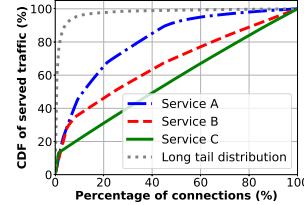


Figure 2: The traffic distribution varies among three different services.

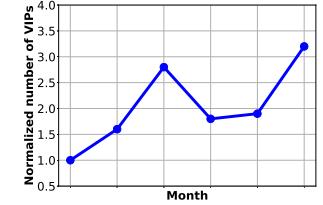


Figure 3: The number of VIPs in a typical LB. It shows a high variation over 6 months.

(where top 10% flows carry 46.3% and 35.5% traffic, respectively) to different extents.

The traffic volume of a service can dynamically change. The traffic volume of an individual service keeps changing independently, with different short-term daily peaks and troughs [21] and long-term uncertainty due to the change in user interest [20]. At any given time in the mixed service traffic, mice flows of one service at peak might consume more bandwidth than elephant flows of another service at the trough, making the overall distribution of their mix unpredictable.

The number of VIPs at a datacenter boundary can change over time. Large service providers keep launching, stopping, and migrating services, driven by various reasons like changes in user interest or business opportunities. Figure 3 reveals a high variation ($3.2\times$) of the number of VIPs served by an LB over 6 months. The dynamic change of services inside the datacenter further makes the mixed VIP traffic at the boundary highly dynamic without any specific distribution.

Based on these observations, we should not rely on any assumption of specific traffic distributions (e.g., long-tail distribution) when designing load balancers at datacenter boundaries for mixed services.

2.3 Accelerating LB with Programmable Switches

Most recent proposals accelerate LBs by realizing hardware Muxes (HMux) [23, 24, 31] with programmable switches, where the RS selection and packet encapsulation are implemented in switch processing pipelines. HMuxes can effectively reduce the number of required servers, which is significant, especially for small/medium-scale datacenters. Nevertheless, using programmable switches as HMuxes suffers from scalability issues on both data and control planes.

Data plane: As widely discussed, switching ASICs cannot support many concurrent flows due to their limited memory sizes [24, 25, 31, 37]. Considering a CT with an entry size of 64 bytes¹ and a typical concurrent flow number of 10M [25, 31],

¹64 bytes/entry is an empirical value for IPv6, including 37 bytes for the

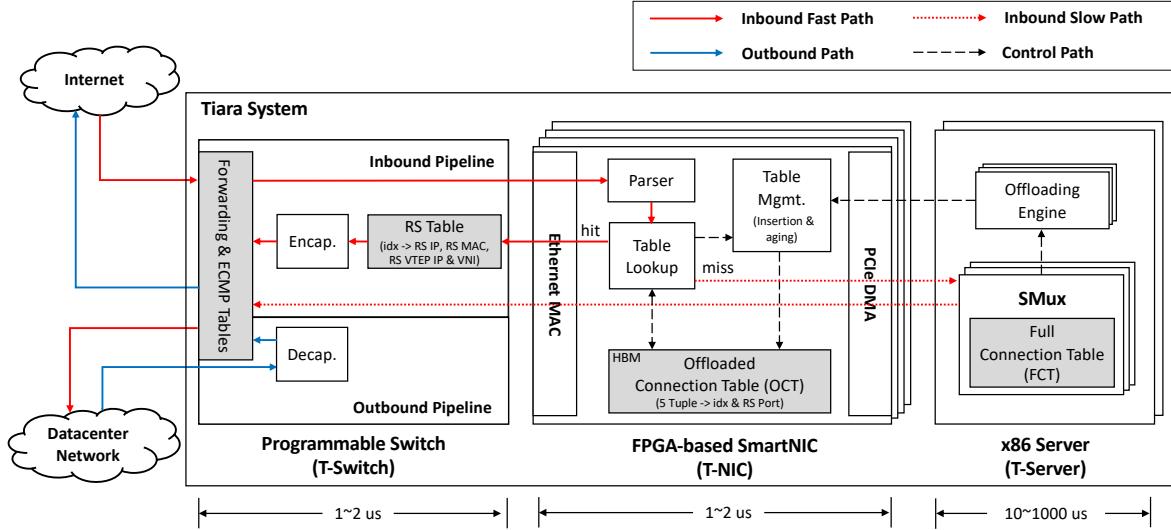


Figure 4: Tiara architecture. Tiara consists of three tiers: T-switch, T-NIC, and T-server. Tiara divides LB into multiple key tasks and matches them respectively to suitable hardware tiers: T-switch for stateless packet encap/decap, T-NIC with HBM for connection lookup and management, and T-server as a last resort.

the CT size is 640 MB. However, modern programmable switches only provide 50-100 MB SRAMs [31]. Moreover, these SRAMs are typically distributed into multiple pipelines, e.g., 15 MB/pipeline. To look up a larger table than a single pipeline’s SRAM size, HMuxes typically use folded pipelines and resubmit a packet to switch pipelines via different physical ports, reducing the available throughput.

Control plane: State-of-the-art programmable switches are slow for entry insertion. For example, a Barefoot Tofino switch can only do $\sim 100K$ insertions per second after our optimizations. We measure the entry insertion overhead. Our result reveals that the top two time-consuming functions are the hash computation and the Cuckoo search algorithm [34]. Our result is similar to those of previous work [16, 31]. The root causes exist in the low-end switch CPU, slow PCIe interconnect between the CPU and the switching ASIC, and the small memory space in the switching ASIC. The first two factors affect the speed of hash computation and operation of offloading. Then the limited memory space forces the switching ASIC to rely on space-efficient Cuckoo hashing for hash collision resolution. The Cuckoo hashing impedes fast insertion by (1) multiple entry movements during collision resolution and (2) incapability of parallelization due to the dependency between two insertions (the previous insertion location may affect the latter one). The above hardware constraints make it difficult for a switch to support $> 1M$ CPS required by production LBs [5].

5-tuple as match key, 18 bytes for RS_IP and RS_Port as action data, and a few bytes for packing and alignment overhead.

3 Tiara Design

We now present Tiara, a novel hardware-accelerated LB architecture, which can support > 1 Tbps traffic, $> 10M$ concurrent flows, and $> 1M$ CPS, without any assumption on traffic patterns.

3.1 Architecture Overview

Tiara is a three-tier architecture as demonstrated in Figure 4. The outermost tier is a programmable switch (T-switch), which sits between the Internet and the datacenter network as a bump in the wire. The second tier is a group of FPGA-based SmartNICs (T-NIC), which act as the HMux jointly with T-switch for LB fast path. The third tier consists of commodity servers (T-server), which host T-NICs and implement SMuxes for LB slow path. The number of T-NICs hosted by a T-server and the number of T-servers behind T-switch are configurable, making the three-tier architecture flexible enough to meet different performance requirements at various datacenter entrances.

The novel idea of Tiara is that it maps different LB tasks into their most suitable devices based on their unique capabilities. In the fast path, Tiara divides the HMux between T-NICs and T-switch. Tiara leverages the large and fast HBM inside T-NIC’s FPGA for memory-intensive *RS selection*. One typical HBM stack comprises 16 256-MB memory channels, and each channel provides ~ 100 million lookups per second (MLPS)². To maximize the accessing performance, we should separate memory accesses to different memory channels. The

²One memory channel provides ~ 100 million random read accesses per second based on our emulation [1].

parallelism among HBM channels is carefully explored to meet memory capacity and throughput requirements of RS selection, which will be discussed in §3.3.1. Meanwhile, Tiara leverages the high performance and programmability properties of T-switch pipelines for throughput-intensive *packet encaps/decap*.

Besides the fast path processing, Tiara instantiates several SMuxes in T-server to act as a backstop for unprocessed traffic. Each SMux maintains a full connection table (FCT) for all inbound flows, and is associated with a T-NIC virtual function with dedicated DMA channels used for packet receiving and sending.

Programmable switch or fixed-function switch. Another option of Tiara’s three-tier architecture demonstrated in Figure 4 is that the programmable T-switch could be replaced by a fixed-function switch that only performs forwarding and ECMP routing. If so, the switch packet processing logic, including RS table, packet encapsulation, and decapsulation, can be moved into T-NICs. We do not choose this option for a few reasons. First, the performance, cost, and power consumption of programmable switches is comparable to that of traditional fixed-function switches [14]. Second, with packet decapsulation implemented in programmable T-switch, the architecture allows outbound traffic to bypass T-NICs (as described in §3.2.2), thus halving the T-NICs bandwidth requirements and the number of required T-NICs. Third, the programmability of T-switch relieves T-NIC implementation. If all fast path functions are implemented in T-NIC, it will increase not only the FPGA size, power consumption, and cost, but also the development time, as programming switches with P4 is easier than programming FPGA with Verilog.

3.2 Control & Data Planes

3.2.1 Control Plane

A typical LB usually includes a centralized controller configuring VIP → RS_IP mappings into Muxes and BGP speakers for VIP announcements. As they are common and well described in previous work [21, 23, 24, 36], we will skip them in this paper and pay more attention to the acceleration-related control flow, i.e., the connection management between software and hardware. Tiara relies on T-servers to make the local control plane decisions, including the CT entry insertion and the entry recycling (connection aging). The powerful CPU prevents inefficient hash computations like that on the switch-based HMux. T-servers use *offloading engines* to offload the entry operations generated by SMuxes, to a specific T-NIC, and each offloading engine is associated with a dedicated DMA channel for entry operations. To efficiently process entry insertion and aging, a few optimizations are made in offloading engines, which will be discussed in §3.3.

Moreover, Tiara integrates many more features like management, telemetry, and fault tolerance in the control plane.

Except for the telemetry, Tiara can support all these functionalities solely in the control plane. Network telemetry requires collecting statistic counters from the data plane, and T-NIC and T-switch can provide them easily without affecting the fast path performance.

3.2.2 Data Plane

Inbound fast path. Upon receiving a packet from the Internet, T-switch distributes it to one of the T-NICs based on ECMP. Then, T-NIC parses the packet header and uses the extracted fields (i.e., 5-tuple) to look up the offloaded connection table (OCT), which maintains up to tens of millions of connections in FPGA HBM and sustains fast lookup. The lookup result from OCT is an LB decision, i.e., a two-tuple {RS_Index, RS_Port}, where RS_Index represents a real server and will be used in later RS table lookup in T-switch. Instead of replacing the RS_Port locally, which will incur checksum computation, Tiara delays this operation to T-switch processing. T-NIC encapsulates the retrieved tuple into a packet metadata header between the Ethernet header and the IP header, and sends back the packet to T-switch. T-switch looks up an RS table and gets the corresponding RS information, including RS_VTEP_IP, RS_MAC, RS_IP, and VNI. Finally, T-switch enforces Port NAT, IP NAT, and VxLAN encapsulation sequentially, and forwards the encapsulated packet to the RS. Since we decouple the RS_Port from the RS table, the number of entries in the RS table is the same as the number of real servers, typically 10K-100K³. Compared to CT, the RS table is relatively stable, updated in second time granularity [36], which is far slower than the entry insertion speed provided by T-switch. Based on these two features, the RS table is achievable in the T-switch SRAMs.

Inbound slow path. When a packet misses in the OCT, the T-NIC uploads it to an SMux via a PCIe DMA channel chosen by *Receive Side Scaling* (RSS). Upon receiving the packet, the SMux looks up the FCT and moves to one of the following two workflows according to the lookup result.

If the packet belongs to an established connection, it will hit in the FCT lookup. SMux retrieves the corresponding {RS_Index, RS_Port}, and further processes the encapsulation for this packet by looking up the RS table locally⁴. Finally, SMux sends the encapsulated packet to the real server (via T-NIC and T-switch). There is a trick on VxLAN source port calculation. Since the source port is calculated by hashing [13], SMux reuses the last 2 bytes of RSS hash value from the T-NIC to avoid duplicate hash computation.

If it is the first packet of a new connection, it will miss in the FCT lookup. SMux makes the LB decision to create a connection entry for this connection and inserts the generated

³A typical datacenter supports thousands of services [19, 36], and each one usually holds 10-100 instances.

⁴In fact, these two tables can fuse into one table.

entry into the FCT. Then, SMux encapsulates the packet and sends it out.

In both cases, SMux will try to insert the corresponding connection entry into OCT. If there are empty slots in the corresponding hash bucket in OCT, the insertion will be successful; otherwise, Tiara will fail the insertion without cache eviction and keep that flow in SMux. We leave the cache eviction policy for the LB connection table as future work.

Outbound path. For outgoing packets, real servers leverage XDP [4] or OVS *Conntrack* [10] to perform SNAT locally. The real servers rewrite source IP with VIP and source ports with *vPort*, and forward the packets in VxLAN encapsulation to T-switch. T-switch further performs packet decapsulation and sends the packets to the Internet. As the only LB operation (i.e., packet decapsulation) for outbound packets is offloaded completely in T-switch, outbound traffic can bypass T-NICs and SMuxes, halving the T-NICs bandwidth requirements.

3.3 Component Design & Optimization

3.3.1 Efficient Hash Table Structure

The hash table design of OCT affects not only lookup performance in hardware but also entry insertion speed in software. We leverage an efficient hash table structure that enables both fast lookup in T-NIC and fast entry insertion in T-server. Specifically, we expect the hash table used in T-NIC should (1) support $O(1)$ and parallel insertions in software and (2) support line-rate lookup in hardware.

We observe that a hash table with fixed-length chaining can satisfy all requirements. First, the insertion complexity of hash chaining is $O(1)$. Second, since the hash computations of different insertion indexes are independent, we can utilize multiple cores in T-server to compute the insertion indexes in a parallel manner. Third, T-NIC can support $O(1)$ lookup by mapping fix-length chains into multiple HBM channels. Last, fix-length hash chaining simplifies hardware design. If using variable-length hash chaining, dynamic memory management is mandatory and unfriendly to hardware implementation.

T-NIC manages OCT using a hash table with fixed-length chaining, as illustrated in Figure 5. Despite the simple structure, determining the proper parameters of the hash table in HBM to achieve both fast lookup and low collision rate is non-trivial. For the hash table with fixed-length chaining, two parameters control the shape of the table: the number of hash indexes (*depth*) and the number of entries at each index (*width*), following that $\text{depth} \times \text{width} = \text{hash table size}$. Given a fixed hash table size, a deeper hash table results in a higher hash collision rate (see analysis in Appendix A), while a wider hash table poses challenges for line-rate lookup on HBM, as the number of parallel HBM memory channels is limited.

Based on the above analysis, T-NIC determines the hash table parameters with a principle that *maximizing the width*

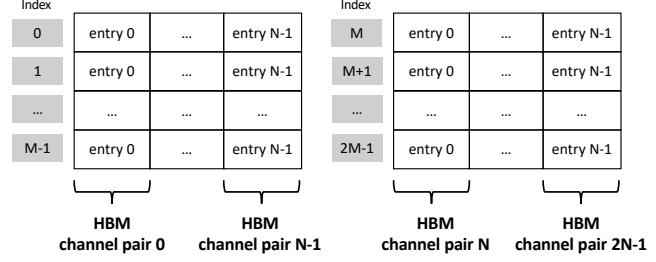


Figure 5: The fixed-length hash chaining design in Tiara OCT, where depth is M and width is N . Each channel pair saves a column of the hash table. For a table lookup, T-NIC can launch multiple parallel accesses of N entries inside $2N$ HBM channels.

while guaranteeing line-rate lookup. Take the FPGA card used in our implementation (§4) as an example. It has two 100GE ports, each requiring 150 MLPS to sustain line rate, and one HBM stack of 16 256-MB ($8M \times 256$ -bit width) memory channels, each providing up to 100 MLPS. We divide 16 channels evenly between two ports so that there are 8 channels to support 100 Gbps traffic. Moreover, the entry size is 512 bits, as discussed in §2.3, so we need to pair two channels for one entry access and construct 4 channel pairs for each port. Given that each channel pair can support 8M entries, there are three candidate hash table structures: $8M \times 4$ (width), $16M \times 2$, $32M \times 1$, where one lookup operation involves 4, 2, and 1 channel pair(s), respectively. However, the lookup performance of the $8M \times 4$ hash table structure is only 100 MLPS (using all channels for one lookup), failing to support the 100 Gbps line rate. Based on the principle, the best hash table structure for one 100GE port is $16M \times 2$ in our FPGA card.

T-NIC relies on the connected T-server to simplify hash collision resolution. When there is a hash collision in the table lookup, T-NIC will forward the packet to the slow path in T-server; when there is a hash collision in the entry insertion, the insertion fails in the offloading engine (§3.3.2), and that flow will be kept in the slow path. As long as the hash collision rate is low (2.6% in theory for 10M flows in the $16M \times 2$ hash table), hash collision does not have significant performance penalty.

3.3.2 Lock-free Offloading Approach

We design a lock-free offloading approach to enable issuing millions of insertion or deletion operations per second from SMuxes to T-NIC, which is required to support $> 1M$ CPS.

In Tiara, SMuxes offload the generated entry operations, including entry insertion and deletion, to T-NICs via offloading engines. Given the multi-channel feature of our PCIe DMA engine, Tiara instantiates a few offloading engines and associates each with a dedicated DMA channel, so that offloading engines can offload entries independently.

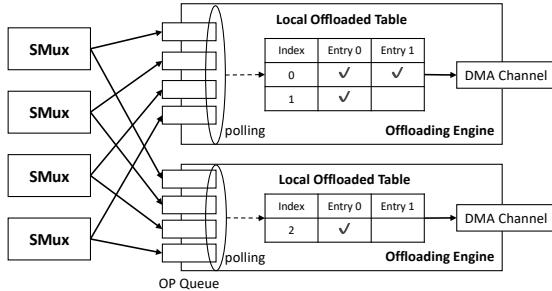


Figure 6: Tiara’s lock-free offloading design.

A straightforward offloading approach introduces locks in two places. The first lock happens when multiple SMuxes are mapped to the same offloading engine with only one OP queue. SMuxes can write their operations to the OP queue only when they retrieve a write lock. The second lock exists when multiple offloading engines insert entries into the same hash index. A lock is required for unavoidable synchronizations on a global OCT, maintained in the server to track the OCT usage among different offloading engines. These two locks prevent us from fully leveraging the parallelism in both the multi-core server and the multi-channel DMA to achieve fast entry offloading.

We design a lock-free offloading mechanism, as shown in Figure 6. First, to realize lock-free entry delivery from SMux to the offloading engine, Tiara sets up an OP queue for each SMux-engine pair. The offloading engine polls OP queues in a round-robin manner to retrieve the offloading operations. Second, Tiara adopts the mapping method based on the entry’s hash index, i.e., *index-to-engine* mapping. Entries inserted into the same place are delivered to the same offloading engine. Consequently, different offloading engines handle entries with different hash indexes, and each offloading engine maintains a local OCT to track the offloaded indexes. Since the local OCTs are disjoint with each other, it is lock-free during the table update.

For each entry operation, offloading engines will notify SMuxes whether the operation is successful or not (an insertion will fail when the corresponding hash bucket is full) via completion queues (not shown in Figure 6).

3.3.3 Lightweight Aging Mechanism

The purpose of this component is to recycle outdated entries in the OCT, i.e., when a connection is disconnected, its related entry in the OCT should be released so that it can be reused for new connections. To realize it, we need to detect the close of connections. One naive method is to use the TCP FIN packet as the signal of the connection close, which can be captured in T-NICs. However, this method fails on abnormal close of TCP traffic and connection-free UDP traffic.

To unify the flow removing process for TCP and UDP, Tiara adopts an entry aging mechanism that removes a flow entry

from the OCT if it is not accessed in a period T . This aging mechanism may kick out connections whose access interval is larger than T by mistake, but those connections can be further processed in the slow path FCT⁵.

The challenge of this aging mechanism is to monitor the accessing states of 10M connection entries periodically with a small memory footprint, minimal performance interference on the data plane, and low CPU overhead.

To address this challenge, T-NIC leverages an accessing bitmap to track connection activities, signals activities to SMuxes, and SMuxes make aging decisions by issuing entry deletion operations based on signals.

T-NIC maintains the accessing bitmap in on-chip SRAMs, using each bit as an indicator for a connection entry. All indicators are reset to 0 at the beginning of every detection period Δ_t ($< T$). An indicator will be marked as active, i.e., set to 1, only if a packet is accessing the corresponding connection entry. As an active signal, the packet header will be sent to an SMux by RSS, ensuring that the same SMux processes both teardown and establishment for a connection. Subsequent packets accessing active connection entries neither change the indicator status nor trigger signaling to SMuxes. In this way, if the connection is active in a detection period, the related SMux will get a signal. If the SMux does not receive any signal for a connection in multiple continuous (T/Δ_t) periods, that connection is considered outdated and should be aged. T-NIC leverages the length of the detection period to control the reporting frequency, which balances the SMux load and the detection precision.

This mechanism is lightweight in three aspects. First, the memory footprint used for tracking connection states in FPGA is minimal, with one bit per connection in the bitmap. Second, as the accessing bitmap is stored in on-chip SRAMs, the aging process will not interfere with HBM lookup in the fast path. Third, given the low signaling frequency (likely to be minutes level), the PCIe and CPU overhead are both low.

4 Implementation

We implement a fully functional prototype of Tiara with one T-switch and one T-server, equipped with one T-NIC through a PCIe Gen3 x16 link. T-switch and T-NIC are connected via 100G Ethernet cables. In the rest of this section, we discuss the implementation details of each component.

4.1 T-switch

We build a P4 prototype of T-switch with a Barefoot Tofino switch, where one pipeline has 12 physical stages, each with 1.25 MB SRAMs and 528 KB TCAMs.

⁵The aging procedure in the FCT is implemented by the SMux, which should provide a longer life cycle for a typical entry compared to the OCT due to its larger memory space.

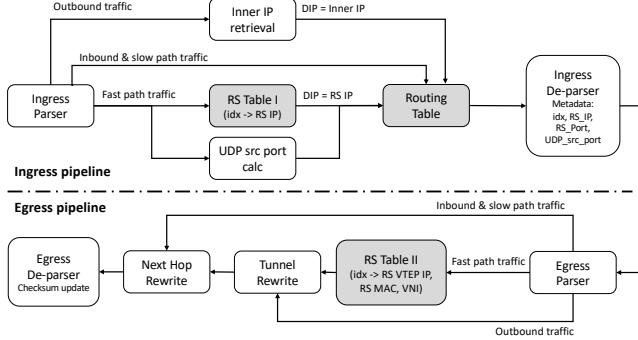


Figure 7: T-switch pipeline implementation.

Metadata					IP	...
ETH	RS_Index (4 B)	RS_Port (2 B)	RSS (2 B)	EtherType = IPv4/IPv6 (2 B)		
EtherType = 0x88B5						

Figure 8: The metadata format.

We modify a baseline switch.p4⁶ to implement the packet processing pipeline, including the RS table, routing tables (forwarding table and ECMP table), and tunnel processing (VxLAN encapsulation and decapsulation). Figure 7 shows an overall pipeline of T-switch. It is worth mentioning that we split the RS table into two parts. The RS table I ($\text{RS_Index} \rightarrow \text{RS_IP}$) exists in the ingress pipeline, where T-switch retrieves RS_IP for routing tables lookup. T-switch postpones the lookup of the rest values in RS table II ($\text{RS_Index} \rightarrow \text{RS_VTEP_IP, VNI, RS_MAC}$), to the egress pipeline. This decoupling helps mitigate resource contention between RS Table and routing tables in the ingress pipeline.

The modified switch.p4 takes 53.85% of SRAMs and 13.19% of TCAMs to implement the pipeline described in Figure 7 with 64K RS table entries, 2K IPv4 addresses, 1K IPv4 prefixes, 1K IPv6 addresses, and 1K IPv6 prefixes.

Recall that, in slow path processing, the VxLAN source port is computed based on the last 2 bytes of RSS value (§3.2.2). To be consistent with the slow path, the fast path in T-switch should compute this field in the same way. However, T-switch pipeline does not support the Toeplitz hash [18, 26, 30] used in RSS computation⁷. To address this issue, T-NIC carries the computed RSS value (last 2 bytes) on packets within an extended metadata header to T-switch (§4.2). T-switch retrieves the hash value from the packet and performs the same computations as SMuxes. In our implementation, the VxLAN source port is computed as followed: $\text{port} = (\text{RSS} \wedge (65535 - 49152)) + 49152$.

4.2 T-NIC

T-NIC is implemented in a Xilinx FPGA-based SmartNIC card, with two 100GE ports and one HBM stack of 16 256MB memory channels. We use Xilinx QDMA IP [11] as the DMA engine. We implement the T-NIC logic described in Figure 4, in System Verilog, including the OCT management and lookup, packet metadata encapsulation, entry aging, and the slow path delivery.

Tiara relies on a metadata header in the packet to pass information between T-NIC and T-switch. Figure 8 shows the format of the metadata header, which is inserted between the Ethernet header and the IP header. The metadata header includes a 4-byte RS_Index, a 2-byte RS_Port, a 2-byte RSS, and a 2-byte EtherType. The field EtherType in the metadata header follows the IEEE 802 standard [6] to indicate the following header type (IPv4 or IPv6). In the Ethernet header, the original EtherType field is changed to 0x88B5, which indicates the next header is private. To avoid the drop of oversized packets caused by inserting the metadata header, we increase the MTU of T-NIC and the corresponding T-switch ports by 10 bytes, i.e., the size of the metadata header.

4.3 T-server

T-server contains 2 Intel(R) Xeon(R) Platinum 8260 CPU. We run SMuxes and offloading engines in one CPU in the same NUMA node as T-NIC without hyper-threading. We build a T-NIC driver as a DPDK [2] PMD and implement the offloading engine on top of it. We leverage an in-house SMux implementation modified from DPVS [3]. The SMux has been deployed over three years, and we make necessary changes to adapt it for the Tiara architecture. The hash computation used in both SMuxes and T-NICs is the CRC32 algorithm.

We optimize the DMA transmission between T-NIC and the PMD. QDMA is a type of Scatter-Gather DMA from Xilinx [11]. For any DMA transaction, it first reads a descriptor from the host to get the physical address of the DMA buffer. The speed of descriptor filling affects the DMA performance. Tiara leverages SIMD instructions provided by Intel processors [7] to accelerate the descriptor filling. For example, we use `_mm_storeu_si128` and `_mm_storeu_si128` to copy the DMA information between the DPDK mbuf and the QDMA descriptor. Moreover, Tiara decouples DMA control channels from data channels to avoid head of line blocking and mutual interference. Tiara guarantees lossless control channels by fine-grained credit control between T-server and T-NIC while remaining data channels to be lossy like conventional NIC data paths.

5 Evaluation

In this section, we use testbed experiments to evaluate the Tiara prototype as described in §4. We first show the micro-

⁶A simplified version can be found at <https://github.com/p4lang/switch>

⁷We follow the standard RSS computation procedure for compatibility

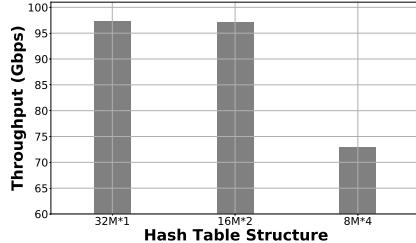


Figure 9: HBM lookup performance on different hash table structures with 10M flows.

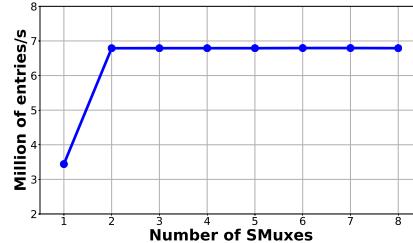


Figure 10: Entry insertion speed of a single offloading engine.

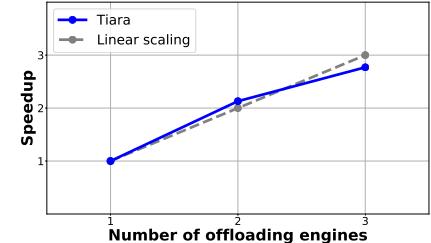


Figure 11: Insertion speedup with multiple offloading engines.

benchmarks to assess the effectiveness of Tiara component designs (§5.1). Then, we measure the end-to-end system performance of Tiara (§5.2). Last, we compare Tiara with existing approaches, i.e., SMux and Silkroad [31] (§5.3). Our results reveal that:

- A T-server with a single T-NIC can provide 200 Gbps throughput with 10M concurrent flows and could scale linearly up to 1.6 Tbps and 80M concurrent flows by hosting 8 T-NICs within a T-server.
- Tiara fast path can provide less than 4 us average latency with small jitter even at line rate.
- Tiara can serve up to 1.8M new connections per second, which is larger than switch-based HMux.
- Tiara is cost-, energy-, and space-efficient, costing 17.4 × less money, consuming 12.8 × less energy, and taking 16.8 × less rack space than SMux, given the same target throughput.

Testbed setup. We leverage the same SMux used in the Tiara slow path as the baseline of software LBs. The Tiara prototype and the baseline are directly connected to the traffic generator using 100 Gbps cables. Test traffic is generated by a hardware traffic generator, sent to the LB (Tiara or baseline), and then routed back to the generator. In this way, we could test the throughput and latency for both Tiara and the baseline.

Traffic. We use a hardware generator to inject synthetic TCP/UDP flows. Since we do not hold any assumption on traffic patterns in Tiara design, the traffic is generated in a random manner.

5.1 Micro-benchmarks

A few micro-benchmarks are designed to evaluate the major component optimizations described in §3.3. Specifically, we evaluate the lookup performance of our hash table design, measure the insertion speed of offloading engines, and test the PCIe overhead incurred by our aging mechanism.

Tiara OCT provides line-rate lookup. We run a benchmark with 10M flows in the OCT, implemented with three candidate hash table structures described in §3.3.1, i.e., 32M×1,

16M×2, and 8M×4. Figure 9 shows the lookup throughput on 10M flows with 128-byte packet size in three candidate hash structures. It reveals that both 32M×1 and 16M×2 structures approach line rate (97.2 Gbps and 97.15 Gbps), but 16M×2 provides a lower theoretic hash collision rate. When the width expands to 4, the throughput drops to 72.9 Gbps since all channels are used for each access at this width. This benchmark is consistent with our analysis in §3.3.1.

Tiara offloading engine achieves fast entry offloading. We randomly generate new flow entries in SMuxes and offload them to T-NIC by offloading engines. Therefore, in this experiment, all offloading operations are entry insertions. Figure 10 demonstrates the offloading speed of a single offloading engine, which is shared among SMuxes. The speed sticks to 6.8M operations per second with more than two SMuxes, which is bounded by the offloading engine. Figure 11 further shows how offloading speed changes with more offloading engines working in parallel. It achieves near linear-scaling with 2.77× speedup when using three offloading engines. The linear scalability of offloading speed makes Tiara able to support a high CPS scenario. For example, the LB in [5] processes 6.9M CPS, requiring at least 13.8M offloading operations (insertion or deletion), which can be supported by two offloading engines, as shown in Figure 11.

Tiara entry aging mechanism incurs negligible overhead. We evaluate the PCIe utilization caused by the aging mechanism, i.e., sending signals (packet headers) to SMuxes via PCIe, with 10M flows and a 1 minute detection period. The average PCIe utilization is less than 0.05%. Given that the control plane and data plane share the same PCIe interface, the low PCIe utilization of Tiara aging mechanism incurs little influence on the data plane.

5.2 Tiara Performance

A complete Tiara system consists of at least one T-switch connected by multiple T-servers, each hosting up to 8 200GE T-NICs. We will show in this section whether such a system could meet the design goals: > 1 Tbps, > 10M concurrent flows, and > 1M CPS, without any assumption on traffic pat-

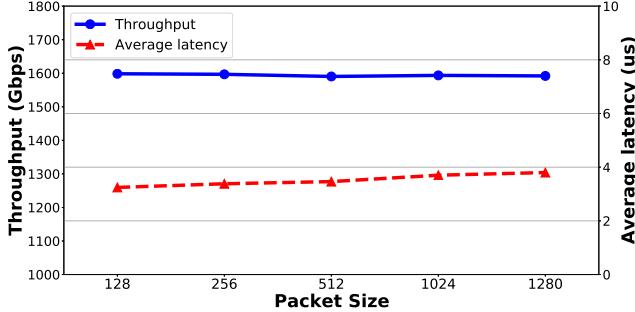


Figure 12: Forwarding performance in Tiara fast path. A T-server with 8 T-NICs achieves up to 1.6 Tbps with less than 4 us latency.

terns.

Throughput and latency. To measure the throughput and latency of Tiara with 10M concurrent flows, we generate traffic consisting of 10M flows and send them to Tiara, which offloads these flows into the fast path.

We first test the performance of Tiara fast path with a single T-NIC. It can achieve the line rate of 200 Gbps and provide an extremely low average latency of less than 4 us, with packet sizes ranging from 128 to 1280 bytes. We further break down the latency distribution in Tiara fast path, which shows about 1 : 1 latency between T-switch and T-NIC.

The throughput and the number of concurrent flows supported in one T-server can scale linearly with the number of T-NICs, as T-NICs plugged in the same server are totally independent of each other, and they share nothing in the fast path processing. As a result, with 8 T-NICs in one T-server, the aggregate throughput of T-server fast path scales linearly to 1.6 Tbps, and the latency remains exactly the same as that of a single T-NIC (i.e., less than 4 us), as shown in Figure 12. Similarly, the number of concurrent flows increases to 80M for a holistic T-server with 8 T-NICs. If the throughput requirement of an LB system is larger than 1.6 Tbps or the flow number requirement is larger than 80M, more T-servers can be connected to the T-switch tier, given the flexibility of this architecture. The aggregate throughput and the flow capacity of Tiara in the fast path can also scale linearly with the number of T-servers, as they are physically independent as well.

CPS. We evaluate Tiara ability to serve new TCP connections by issuing HTTP transactions, including a TCP connection establishment, an HTTP GET request, an HTTP response (by-passing LB), and a closure of TCP connection. We gradually increase the target CPS in 0.1M granularity at the generator to find the maximum available CPS that the target LB can serve all the incoming requests. The result reveals that Tiara can support up to 1.8M CPS (bounded by SMux), which is higher than our goal (i.e., 1M CPS).

Resilience to traffic patterns. By leveraging the large capacity of FPGA HBM for the connection table, almost all

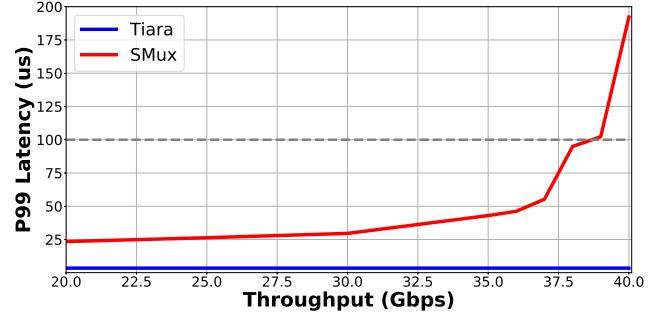


Figure 13: Latency-bounded throughput. With the tail (P99) latency bound of 100us, Tiara can achieve 200Gbps per T-NIC and 1.6Tbps per T-server with 8 T-NICs. However, since SMux suffers from high jitter when the load increases, the maximum latency-bounded throughput of SMux is 38Gbps.

flows can be offloaded to the fast path in Tiara as long as the number of concurrent flows is less than 10M per T-NIC and 80M per T-server, which is true in most cases as we observed at our datacenter boundaries. As a result, Tiara is insensitive to traffic patterns, and it keeps consistent high throughput and low latency on different traffic patterns.

5.3 Tiara vs. Existing Approaches

In this section, we compare Tiara with existing approaches (the SMux baseline, Silkroad [31]) in terms of performance and efficiency. The results are summarized in Table 1.

Performance. SMux suffers from high latency and jitter when the traffic load is heavy [33] due to high CPU utilization and cache misses. High latency and jitter will adversely impact the user’s network experience. Therefore, we use “latency-bounded throughput” as the metric to compare SMux and Tiara more fairly. Given that the end-to-end latency from Internet users to datacenter services could be as low as a few milliseconds [35, 39], we should bound the tail latency of LB to the sub-millisecond level to minimize its impact on the user’s network experience. In this experiment, we run SMux on 16 cores of a server with the same configuration as T-server (§4.3), except that the SMux server is equipped with a 100 Gbps Mellanox ConnectX-5 NIC rather than T-NICs. We set the bound of LB P99 latency to 100 us and compare the latency-bounded throughput with the packet size of 512 bytes between Tiara and SMux. Figure 13 shows the P99 latency of Tiara and SMux, respectively, with different throughputs. For the Tiara fast path, P99 latency is consistently below 4 us at throughput up to 200 Gbps per T-NIC, while SMux P99 latency breaks the 100 us bound when the throughput is higher than 38 Gbps. Therefore, we consider 38 Gbps as the maximum latency-bounded throughput of the baseline SMux. In Tiara, the latency-bounded throughput of a single T-server with 8 T-NICs is 1.6 Tbps, 42.1× higher than SMux (38 Gbps), and its P99 latency (4 us) is 25× lower than

	Throughput	P99 lat.	CPS	CT size*	Cost efficiency	Energy efficiency	Space efficiency
SMux	38 Gbps	100 us	1.8M	~100 GB	4.75 Gbps/(cost unit)	76 Mbps/Watt	19 Gbps/U
Silkroad**	1.6 Tbps	< 2 us	200K	100 MB	457.14 Gbps/(cost unit)	2909.1 Mbps/Watt	1600 Gbps/U
Tiara	1.6 Tbps	< 4 us	1.8M	4 GB	82.05 Gbps/(cost unit)	969.7 Mbps/Watt	320 Gbps/U

Table 1: Performance and efficiency comparison among different LBs. *Since the connection table (CT) compression in Silkroad is orthogonal to Tiara and can be applied in any architecture, we use the CT size as the metric to compare the data plane scalability of different architectures. **The Silkroad paper does not report throughput and tail latency explicitly, and we use the same throughput and latency results as T-switch to simplify comparison.

SMux (100 us).

Silkroad achieves comparable high throughput and low latency as Tiara, as most connections are processed in the hardware fast path in both solutions. However, Silkroad is less scalable in both control and data paths than Tiara. Silkroad leverages the embedded management CPU in switch for connection creation and offloading, thus expecting only 200K CPS [31]. Tiara achieves 1.8M CPS, 9× higher than Silkroad, thanks to the optimizations in the control plane of Tiara. Silkroad stores the connection table in the switch’s limited on-chip SRAMs. Despite compression with hash digest, the connection table is still bounded by the on-chip SRAM size, i.e., 50-100 MB in modern switching ASICs. Tiara leverages 4 GB HBM in modern FPGA, increasing the connection table size in the fast path by orders of magnitude compared to Silkroad.

Efficiency. In this section, we quantify and compare the efficiency of SMux, Silkroad, and Tiara, in terms of cost efficiency (performance per dollar), energy efficiency (performance per watt), and space efficiency (performance per rack unit).

- **Cost efficiency.** As the concrete cost numbers of T-NIC, T-switch, and T-server used in the Tiara prototype are confidential, we normalize them to 1, 3.5, and 8, respectively. With these cost units, the normalized system costs of SMux, Silkroad, and Tiara are 8, 3.5, and 19.5 (=3.5+1*8+8), respectively. Given these normalized system costs and the throughput data shown in Table 1, the cost efficiency of these three approaches will be 4.75 Gbps/(cost unit), 457.14 Gbps/(cost unit), and 82.05 Gbps/(cost unit), respectively.
- **Energy efficiency.** According to hardware datasheets, T-NIC, T-switch, and T-server used in the Tiara prototype consume 75 Watt, 550 Watt, and 500 Watt power, respectively. Based on these power consumption and throughput data, the energy efficiency of SMux, Silkroad, and Tiara will be 76 Mbps/Watt, 2909.1 Mbps/Watt, and 969.7 Mbps/Watt, respectively.
- **Space efficiency.** The server used in SMux is 2 rack-unit (i.e., 2U) high, the switch used in Silkroad is 1U high, and the entire Tiara system is 5U high, as it includes a 1U T-switch and a 4U T-server hosting 8 T-NICs. Based on these

heights and throughput data, the space efficiency of SMux, Silkroad, and Tiara will be 19 Gbps/U, 1600 Gbps/U, and 320 Gbps/U, respectively.

- **Tiara vs. SMux in efficiency.** The cost, energy, and space efficiency of Tiara are 17.4×, 12.8×, and 16.8× higher than those of SMux, respectively. In other words, given the same target throughput, Tiara costs 17.4× less money, consumes 12.8× less energy, and takes 16.8× less rack space than SMux. All these efficiency advantages of Tiara over SMux come from hardware acceleration, as suitable hardware (i.e., FPGA and programmable switch in Tiara) is fundamentally much more efficient than x86 servers in network packet processing.
- **Tiara vs. Silkroad in efficiency.** As we can see from Table 1, the switch-only solution in Silkroad outperforms Tiara in all efficiency metrics. This is expected as Silkroad only leverages a switch, which is fundamentally more cost-, energy- and space-efficient than FPGA and x86 in network packet processing. However, as we discussed in the above section, the efficiency of Silkroad comes at the cost of lower CPS and smaller connection tables due to switch inherent scalability limitations. Compared to Silkroad, Tiara strikes a better balance between efficiency and scalability. Furthermore, the switch-only solution may not be that practical in traffic scenarios with a large number of connections, where Silkroad suggests operators combine its switch with an SMux for the slow path [31]. With this hybrid setting (switch + server), the efficiency of Silkroad will become similar to Tiara, but its scalability in hardware is still lower than Tiara.

One more option to further improve the efficiency of Tiara is to bake its implementation into a custom ASIC, which makes it as efficient as the Silkroad switch-only solution and as scalable as current Tiara. However, a custom ASIC incurs a significant NRE (non-recurring engineering) cost. Without a big enough volume to amortize the NRE, the cost efficiency of custom ASIC is worse than that of current Tiara design. As the performance of a single T-server is already high enough (up to 1.6 Tbps), we do not necessarily need a large number of T-servers to load-balance Internet traffic in even hyper-scale datacenters. Therefore, the design choice of using FPGA rather than custom ASIC in Tiara is justified in this context.

6 Related Work

Memory enhanced switches: eXtra Large Table (XLT) [17] enhances programmable switches with FPGA + DRAM complexes to support large tables. It works well when all rule/flow tables are stored in DRAM, and the switch and FPGA can handle all data plane processing entirely. However, that is not the case for stateful load balancers discussed in this paper. Despite large DRAM, packet lookup may still miss in XLT FPGA due to hash collision or first packet processing for new connections, but how to handle these exceptions is unclear.

TEA [25] extends switching ASIC memory virtually by utilizing the host DRAM via RDMA. However, looking up a table at the remote memory prevents switches from line-rate processing. TEA relies heavily on traffic locality that caches hot traffic in the on-chip SRAMs to preserve high throughput. Otherwise, its performance approaches the server-based lookup table, as demonstrated in its experiment (TEA with and without cache). Moreover, TEA shares the same scalability issue on the control plane as other programmable switches.

Layer-4 load balancing: There have been continuous efforts on layer-4 load balancing. In general, two LB categories are explored: stateful LBs that keep the per-connection state at Muxes and stateless LBs that do not maintain any per-connection state.

Ananta [36] and Maglev [21] are two proposed software stateful LBs with a series of packet processing optimizations, including batch processing, poll mode NIC driver, and zero-copy operations. Despite these optimizations, the packet forwarding throughput on a single server is still limited, so that they need a large number of servers to support terabits per second traffic.

Duet [24] and Rubik [23] accelerate Ananta with commodity switches in a stateless style. They store the VIP-to-DIP (RS_IP) mapping in switch on-chip SRAMs as an ECMP table. To support large-scale mapping rules, they leverage the tail distribution in VIP traffic to configure the heavy-hitting rules on switches while processing the rest in the software.

Beamer [33] is a recently proposed stateless LB. It relies on hash functions on the switch to proceed fast real server selection and uses "daisy chaining" techniques to mitigate the PCC violations. The "daisy chaining" requires real servers to redirect unexpected packets. However, it is empirically impractical to modify the service servers. Moreover, stateless LBs can only provide suboptimal workload balancing due to the nature of hash functions as described in [16].

Silkroad [31] is the most related work, which accelerates stateful LB with programmable switches. It faces the same problems as mentioned in §2.3, but it only focuses on addressing the data plane scalability issue with on-chip SRAMs. Silkroad stores a hash digest of a connection instead of the 5-tuple in the connection table, which reduces the key size

of each connection from dozens of bytes to 16 bits. Such compression technique scales to support millions of concurrent flows. However, Silkroad will suffer from throughput degradation due to pipeline folding for those switches that distribute their SRAM resources in multiple pipelines.

Cheetah [16] aims to design a high-speed LB for both stateless and stateful manners. One of its contributions is to solve the entry insertion inefficiency problem in stateful LB by storing unused hash indexes in a connection stack. For every new coming connection, Cheetah pops an index from the connection stack and inserts the connection entry into the hash table with the retrieved index. This index, encoded in the packet header as a cookie, is carried by the connection in the following packets. The change on the packet header requires modifications on services' client sides. This requirement prevents Cheetah from deploying on large-scale datacenters with hundreds and thousands of services.

Component design & optimization: Some techniques used in the component design and optimization in Tiara have been extensively studied. Tong et al. [41] propose a high-throughput hash table structure with the idea of fixed-length hashing in FPGA DRAM. Mogul et al. [32] eliminate the livelock by a polling-based mechanism, and Kuperman et al. [27] match each net device TX queue to a hardware send queue to avoid spin-lock contention. Ross [38] splits tables into different cores in a multi-core database system to reduce the synchronization cost. The SmartNIC used in Azure [22] periodically reports flow states to the software, which allows the software manager to age the inactive flows. Our contribution is to integrate those techniques to achieve the design goals of Tiara.

7 Conclusion

Tiara is a novel hardware acceleration architecture for stateful load balancers. It simultaneously provides high throughput, low latency, high scalability, and high efficiency by mapping different LB tasks into their most suitable hardware and carefully designing and optimizing a few key components. Although we only show Tiara's capabilities to accelerate stateful load balancers in this paper, we believe this architecture is generic for network function acceleration and can be explored in the future in more gateway scenarios, such as DDoS protection and firewall.

Acknowledgments

We thank our anonymous reviewers and shepherd Anuj Kalia for their insightful comments. We also thank Naiqian Zheng, Kaicheng Yang, and Yuxuan Gao for their support of the project. The work of Chaoliang Zeng, Zilong Wang, and Kai Chen was supported in part by a ByteDance Research Collaboration Project and the Hong Kong RGC TRS T41-603/20-R and GRF 16215119.

References

- [1] Axi high bandwidth memory controller v1.0. https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf.
- [2] Dpdk. <https://www.dpdk.org/>.
- [3] Dpv is a high performance layer-4 load balancer based on dpdk. <https://github.com/iqiyi/dpv>.
- [4] express data path. <https://www.iovisor.org/technology/xdp>.
- [5] High-performance dpdk-based server load balancing for alibaba singles' day shopping festival. <https://www.alibabacloud.com/blog/593984?spm=a2c5t.11065265.1996646101.searchclickresult.289b2f05911A1a>.
- [6] Ieee 802 numbers. <https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml>.
- [7] Intel intrinsics guide. <https://software.intel.com/sites/landingpage/IntrinsicsGuide>.
- [8] Katran: A high performance layer 4 load balancer. <https://github.com/facebookincubator/katran>.
- [9] Load balancing 101: Nuts and bolts. <https://www.f5.com/services/resources/glossary/load-balancer>.
- [10] Ovs conntrack. <https://docs.openvswitch.org/en/latest/tutorials/ovs-conntrack/>.
- [11] Qdma subsystem for pci express. <https://www.xilinx.com/products/intellectual-property/pcie-qdma.html>.
- [12] Unveiling the networks behind the 2018 double 11 global shopping festival. <https://www.alibabacloud.com/blog/594167?spm=a2c5t.11065265.1996646101.searchclickresult.289b2f0575gg5Z>.
- [13] Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. <https://tools.ietf.org/html/rfc7348>.
- [14] Anurag Agrawal and Changhoon Kim. Intel tofino2—a 12.9 tbps p4-programmable ethernet switch. In *HCS 2020*.
- [15] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. Balancing on the edge: Transport affinity without network state. In *NSDI 2018*.
- [16] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *NSDI 2020*.
- [17] Curt Beckmann, Ramkumar Krishnamoorthy, Han Wang, Andre Lam, and Changhoon Kim. Hurdles for a dram-based match-action table. In *ICIN 2020*.
- [18] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. Umac: Fast and secure message authentication. In *CRYPTO 1999*.
- [19] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM 2012*.
- [20] Alan Edelman. Akamai technologies: A mathematical success story. In *SIAM News 1999*.
- [21] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jin-nah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *NSDI 2016*.
- [22] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *NSDI 2018*.
- [23] Rohan Gandhi, Y Charlie Hu, Cheng-Kok Koh, Hongqiang Harry Liu, and Ming Zhang. Rubik: unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *ATC 2015*.
- [24] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *SIGCOMM 2014*.
- [25] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *SIGCOMM 2020*.
- [26] Hugo Krawczyk. Lfsr-based hashing and authentication. In *CRYPTO 1994*.
- [27] Yossi Kuperman, Maxim Mikityanskiy, and Rony Efraim. Hierarchical qos hardware offload (htb).
- [28] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *SOSP 2017*.

- [29] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *SIGCOMM 2016*.
- [30] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. In *TCS 1993*.
- [31] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *SIGCOMM 2017*.
- [32] Jeffrey C Mogul and KK Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *TOCS 1997*.
- [33] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *NSDI 2018*.
- [34] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Journal of Algorithms 2004*.
- [35] Fabio Palumbo, Giuseppe Aceto, Alessio Botta, Domenico Ciuonzo, Valerio Persico, and Antonio Pescapé. Characterization and analysis of cloud-to-user latency: the case of azure and aws. In *CN 2021*.
- [36] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. In *SIGCOMM 2013*.
- [37] Kun Qian, Sai Ma, Mao Miao, Jianyuan Lu, Tong Zhang, Peilong Wang, Chenghao Sun, and Fengyuan Ren. Flexgate: High-performance heterogeneous gateway in data centers. In *APNet 2019*.
- [38] Kenneth A Ross. Multicore processors and database systems: The multicore transformation. In *Ubiquity 2014*.
- [39] Ao-Jan Su, David R Choffnes, Aleksandar Kuzmanovic, and Fabian E Bustamante. Drafting behind akamai: Inferring network conditions based on cdn redirections. In *TON 2009*.
- [40] Ao-Jan Su and Aleksandar Kuzmanovic. Thinning akamai. In *SIGCOMM 2008*.
- [41] Da Tong, Shijie Zhou, and Viktor K Prasanna. High-throughput online hash table on fpga. In *IPDPS 2015*.
- [42] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. Fpga-accelerated compactations for lsm-based key-value store. In *FAST 2020*.

A Analysis on Hash Collision

Suppose there are n random entries inserted into a hash table with width w and depth d . The probability that any i entries are hashed to the same index is:

$$p(i) = C_n^i \left(\frac{1}{d}\right)^i \left(1 - \frac{1}{d}\right)^{n-i} \quad (1)$$

The probability for any indexes that hold $0 \sim w$ entries is:

$$p(\text{num} \leq w) = \sum_{i=0}^w C_n^i \left(\frac{1}{d}\right)^i \left(1 - \frac{1}{d}\right)^{n-i} \quad (2)$$

For all indexes, this probability becomes:

$$p(\text{num} \leq w)_{\text{all}} = \left(\sum_{i=0}^w C_n^i \left(\frac{1}{d}\right)^i \left(1 - \frac{1}{d}\right)^{n-i}\right)^d \quad (3)$$

Therefore, the probability for all indexes that exist at least once collision, i.e., holding more than w entries, is:

$$p(\text{num} > w)_{\text{all}} = 1 - \left(\sum_{i=0}^w C_n^i \left(\frac{1}{d}\right)^i \left(1 - \frac{1}{d}\right)^{n-i}\right)^d \quad (4)$$

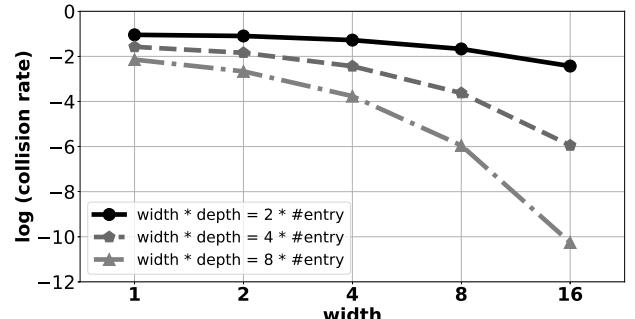


Figure 14: The numerical simulation on collision rates in different widths and depths with $\# \text{entry} = 32768$. The collision rates are shown in the log scale.

To get an intuitive relationship between the collision rate and the width, we conduct a numerical simulation on different settings based on Equation 4. The results are demonstrated in Figure 14, and show that given a fixed hash space ($> n$), a larger width results in a lower hash collision rate.

Scaling Open vSwitch with a Computational Cache

Alon Rashelbach, Ori Rottenstreich, Mark Silberstein
Technion

Abstract

Open vSwitch (OVS) is a widely used open-source virtual switch implementation. In this work, we seek to scale up OVS to support hundreds of thousands of OpenFlow rules by accelerating the core component of its data-path - the packet classification mechanism. To do so we use NuevoMatch, a recent algorithm that uses neural network inference to match packets, and promises significant scalability and performance benefits. We overcome the primary algorithmic challenge of the slow rule update rate in the vanilla NuevoMatch, speeding it up by over *three orders of magnitude*. This improvement enables two design options to integrate NuevoMatch with OVS: (1) using it as an extra caching layer in front of OVS’s megaflow cache, and (2) using it to completely replace OVS’s data-path while performing classification directly on OpenFlow rules, and obviating control-path upcalls. Our comprehensive evaluation on real-world packet traces and ClassBench rules demonstrates the geometric mean speedups of $1.9\times$ and $12.3\times$ for the first and second designs, respectively, for 500K rules, with the latter also supporting up to 60K OpenFlow rule updates/second, by far exceeding the original OVS.

1 Introduction

Open vSwitch (OVS) [22] is one of the most popular software switches used by cloud providers to implement software-defined networks [1, 8, 23]. As part of its main tasks, OVS classifies packets according to a set of match-action tuples, i.e., OpenFlow rules dynamically installed by the network controller. To achieve high throughput, OVS adopts the fast/slow path separation principle: the majority of the packets are classified in the fast *data-path*, which maintains a *megaflow cache* optimized for speedy matching. Upon a miss, OVS invokes the slower upcall into a *control-path*, which populates the megaflow cache with tuples called megaflows.

Unfortunately, OVS suffers from two primary scalability issues. First, the megaflow cache becomes slower as the number of megaflows in it grows. Our experiments (§3) show that

with 500K megaflows, OVS is about an order of magnitude slower than with 1K megaflows. Importantly, the cache might hold a large number of megaflows even if the number of the original OpenFlow rules is small. This is because when OVS populates the cache, it transforms the relevant OpenFlow rules into a set of non-overlapping megaflows [22]. As a result, the OpenFlow rules might get fragmented; under certain common traffic patterns, this fragmentation leads to a dramatic increase in the number of megaflows in the cache [3, 4].

The second problem is the performance degradation that occurs when new rules are inserted into OVS by a network controller. We observe (§3) that the throughput might be affected significantly even when adding only a few dozens of new OpenFlow rules at a time. The main reason stems from the need to enforce the non-overlapping property of megaflows, which might cause OVS to remove existing megaflows, leading to slow path upcalls. Clearly, the problem gets worse in systems with frequent rule updates.

In this work, we seek to overcome these OVS limitations. Our key idea is to leverage the recently published algorithm for packet classification, called *NuevoMatch* [26, 27], which was shown to significantly outperform state-of-the-art alternatives when scaling to a large number of OpenFlow rules. NuevoMatch uses shallow neural networks comprising a *Range-Query Recursive Model Index* (RQ-RMI) to learn the distribution of the rules. The rule lookup is translated into neural-network inference that replaces the traditional index data structure traversal. Upon an update, new rules are first added to a slow-path *remainder* classifier, and the model is periodically retrained to incorporate them in the fast path. Thus, the RQ-RMI model serves as a *computational cache* for the remainder, while retraining the model is equivalent to filling that cache. The scalability of NuevoMatch follows from its small memory footprint and efficient use of CPU hardware, which together enable fast execution on modern CPUs [26].

However, our initial attempts to integrate OVS and NuevoMatch revealed one critical limitation of the original algorithm: its inability to accommodate fast updates. When rules are modified, NuevoMatch must *retrain the RQ-RMI model*

from scratch on the updated rule-set, in order to reach its full performance potential. Unfortunately, RQ-RMI training time is too long and cannot support the required update rate, particularly with a large number of OpenFlow rules as targeted by our work. Our analysis (§3) shows that the NuevoMatch training rate is *orders of magnitude* slower than the one necessary to achieve its promised performance benefits.

We tackle this challenge by introducing *NuevoMatchUP* which extends the original NuevoMatch training algorithm and improves the training rate by over *three orders of magnitude*. Thus, it requires only a few milliseconds to train tens of thousands of rules, and about one second for 500K rules, thereby paving the way to the practical integration of computational cache into OVS.

We consider two design options for integrating NuevoMatchUP with OVS. The first design, *OVS with computational cache* (OVS-CCACHE), targets the scalability of the megaflow cache by accelerating it with NuevoMatchUP. ovs-CCACHE achieves higher throughput than the original design, but unfortunately inherits the low rule update performance. To support fast updates, we introduce *OVS with computational flows* (OVS-CFLOWS), which leverages the power of NuevoMatchUP to efficiently match complex OpenFlow rules and obviates the need for the megaflow cache and fast-slow path separation of the original OVS. This change eliminates the key bottleneck that restricts the rule update rates in the original OVS.

We comprehensively evaluate OVS-CCACHE and ovs-CFLOWS using real-world CAIDA [2] and MAWI [37] traces, and the standard ClassBench-generated rule-sets [32]. OVS-CCACHE improves the megaflow cache performance, achieving the end-to-end geometric mean speedups of $1.5\times$, and $1.9\times$ for 100K, and 500K OpenFlow rules, respectively.

OVS-CFLOWS sidesteps the control-path limitations and is thus significantly faster, with the end-to-end geometric mean speedups of $2.6\times$, $8.5\times$, and $12.3\times$ for 1K, 100K, and 500K OpenFlow rules, respectively. Moreover, OVS-CFLOWS handles more than 60K OpenFlow rule updates/second.

These results demonstrate the first practical use of RQ-RMI models in a production packet processing system, and show their ability to improve throughput and scalability.

2 Background

We explain the relevant details about the operation of Open vSwitch (OVS) [22, 23], and describe the NuevoMatch algorithm [26] for packet classification.

2.1 Open vSwitch

Open vSwitch (OVS) is a popular open-source virtual switch that supports industry standard OpenFlow protocols. OVS determines which action to apply on each packet according to the OpenFlow rules installed by the network controller.

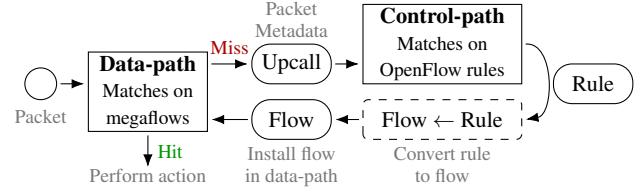


Figure 1: Fast/slow path separation in OVS.

This task is known as *packet classification*, and has been extensively studied [6, 10, 18, 19, 26, 28, 30, 35, 39].

Matching a rule. In its simplest form, a *rule* is a boolean predicate parametrized by one or more fields in the packet header (e.g., IP address, IP protocol). If a predicate is true for a given packet (the rule *matches*), an *action* associated with the rule is invoked to process the packet. An action is an operation to apply to the packet (e.g., forward to port or drop). A packet may match several *overlapping* rules, but only the one with the highest priority is selected.

Control-/data- path. OVS is split into data- (fast) and control- (slow) paths (Figure 1). All OpenFlow rules are installed and maintained in the control-path. The data-path, on the other hand, uses a *megaflow cache* to achieve high processing rates.

The megaflow cache holds non-overlapping rules called *megaflows*, generated by the control-path from the installed OpenFlow rules. Specifically, whenever the data-path encounters a packet that does not match any previously installed megaflow, it performs an *upcall* to the control-path, which in turn finds the relevant OpenFlow rule and converts it into a megaflow. Future packets with the same header fields will not require upcalls unless the megaflow is removed. OVS ensures the correctness of the matching process with the megaflow cache, terminating lookup after a hit in it. To achieve that, OVS tracks all modifications to the OpenFlow rules in the control-path. In particular, it might need to invalidate previously installed megaflows when new OpenFlow rules are added. As we show in §3, these operations might significantly affect OVS’s performance.

In addition to the megaflow cache, OVS often activates a short-term exact-match cache (EMC) in front of it. The EMC can be helpful with high-locality traffic.

Megaflow cache implementation. The megaflow cache uses the *Tuple Space Search* (TSS) [30] algorithm for packet classification, as follows. Megaflows with the same mask m are stored in the same hash table H_m , with masked flow keys as entries. Given a packet header h , the megaflow cache iterates over all hash tables to find an entry that matches h (i.e., the masked header equals to the masked key). The lookup latency increases linearly with the number of hash tables traversed.

OVS’s data-path can run either in the user-space using DPDK [25], or as a dedicated Kernel module. In this paper we use the DPDK version for its higher performance [34].

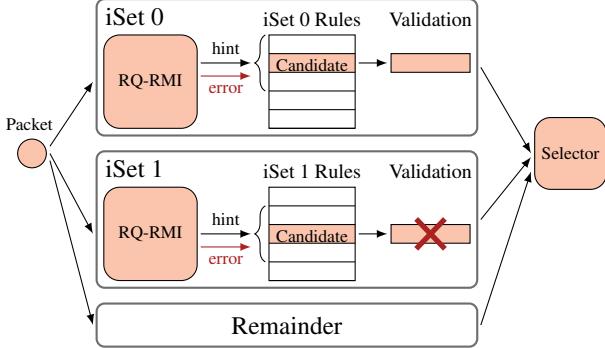


Figure 2: NuevoMatch algorithm [26], RQ-RMI inference provides hints to find the matching rule (details in §2.2).

Size Category	SC 0	SC 1	SC 2	SC 3
# Input Rules	< 10^3	$10^3 - 10^4$	$10^4 - 10^5$	> 10^5
# Neural Nets	5	21	133	265 or 521

Table 1: RQ-RMI model size (number of neural nets) for different number of rules to index (values taken from [26]).

2.2 NuevoMatch Classification Algorithm

NuevoMatch (NM) is a new class of packet classification algorithms that leverage neural nets to scale to many rules [26].

Figure 2 presents the main components of the algorithm. NM partitions a given set of rules into several independent subsets (iSets), such that each iSet s has a header field h_s in which its rules do not overlap. The fraction of an iSet's rules out of all rules is called the *iSet's coverage*. In practice, two iSets are often sufficient to cover more than 90% of the rules for large enough rule-sets [26]. Rules that do not fit in any of the iSets are handled by a *remainder classifier*, which can be implemented by any other packet classification technique.

For each iSet s , NM trains a hierarchical model called *Range-Query Recursive Model Index* (RQ-RMI) which consists of multiple shallow neural-nets. RQ-RMI learns the distribution of ranges represented by the rules and outputs the *estimated index* of the matching rule within an array. At the inference time, this estimation is used as a starting index to *search* for the matching rule within the array. Crucially, the RQ-RMI training algorithm guarantees a tight bound on the maximum error of the estimated index, which in turn bounds the search and ensures lookup correctness. During the search, the candidate rules are *validated* by matching over *all* fields of the incoming packet. Finally, the highest priority rule is selected out of all the matching rules from all the iSets and the remainder.

The number of neural nets (NNs) in an RQ-RMI model depends on the number of rules it indexes. The original paper suggests four RQ-RMI size categories, reported in Table 1. The larger the model, the longer it takes to train it. However

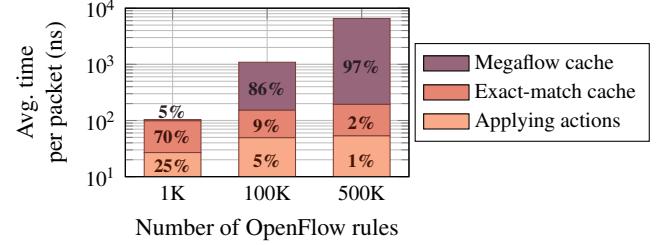


Figure 3: Breakdown of packet processing times in the data-path for different number of OpenFlow rules.

smaller models would fail to achieve the target error bound guarantees and would result in a slower lookup. Thus, there is a fundamental trade-off between the lookup latency and the training time. NuevoMatchUP changes the way RQ-RMI is constructed to modify this trade-off, allowing a much faster training with negligible degradation in the lookup latency.

3 Motivation

We analyze OVS's scalability bottlenecks and highlight the potential benefits of using faster packet classification.

For the analysis we use the same setup and workloads as described in §7. In particular, we generate 36 ClassBench OpenFlow rule-sets (12 application types of three size categories each: 1K, 100K, 500K rules), and evaluate the throughput by replaying Caida-short packet trace (100M packets).

Does OVS get slower with more rules? We compare the throughput with 1K rules vs. the throughput with 100K and 500K rules, separately for each ClassBench application type. We observe that the geometrical mean *slowdown* for 100K and 500K rules vs. 1K rules is 5.8× and 9.1×, respectively.

Takeaway 1: OVS does not scale well to a large number of OpenFlow rules.

Where is the bottleneck in the data-path? We analyze the average processing time of a packet in the OVS data-path while varying the number of OpenFlow rules across all the rule-sets. Figure 3 shows that packets spend the majority of time in the megaflow cache, i.e., 86% and 97% of the CPU time on average, for 100K and 500K rules respectively.

Takeaway 2: OVS megaflow cache becomes the main data-path performance bottleneck as the number of OpenFlow rules increases.

Are the control-path upcalls the primary bottleneck? Misses in the megaflow cache trigger upcalls into the control-path. The frequency of the upcalls is hard to predict; it depends on the interplay between the rule-set and the traffic pattern [3, 4]. Unfortunately, frequent upcalls cause major throughput drop. For example, Figure 4 shows the throughput and the rate of deletions and upcalls, sampled every 100ms, for a 100K rules (rule-set number 2 in §7). The higher the

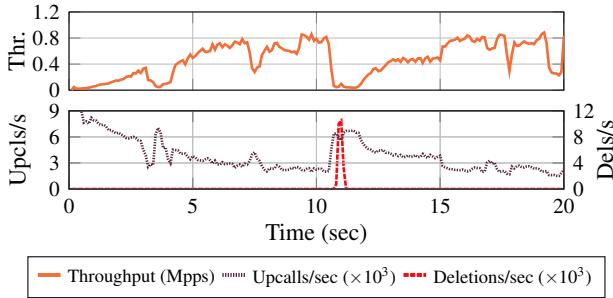


Figure 4: OVS throughput is affected by control-path upcalls.

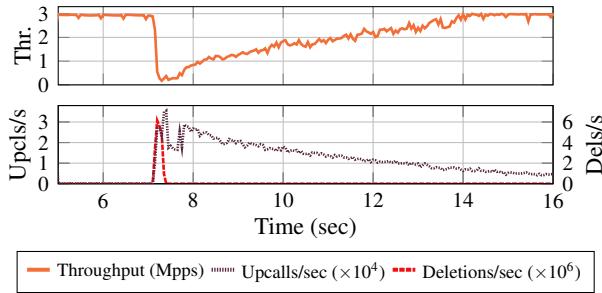


Figure 5: Insertion of new OpenFlow rules: the throughput drops at $t = 7$ s when 60 new OpenFlow rules are added to 500 existing ones. Note the coinciding peak in the deletion rate from the megaflow cache and the subsequent increase in the number of upcalls.

number of upcalls, the lower the throughput. Similarly, the performance drop is observed due to deletions, triggered by the periodic megaflow cache cleanup of idle flows. For other rule-sets the behavior is similar.

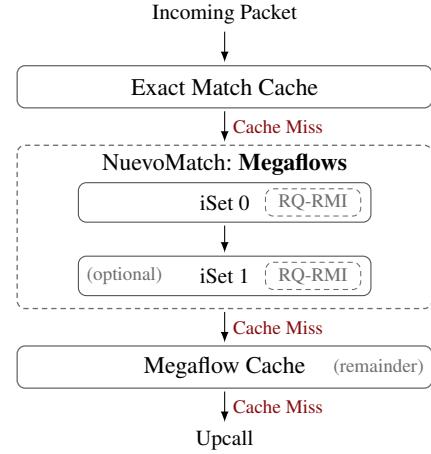
Takeaway 3: frequent upcalls are detrimental to performance.

Impact of OpenFlow rule updates. OVS might experience a sharp drop in throughput when OpenFlow rules are modified.

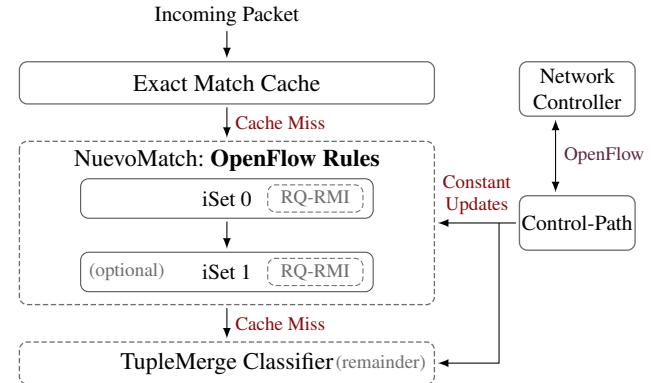
To show that, we install 500 OpenFlow rules in the beginning and update 60 rules at time $t = 7$. Figure 5 shows the results. The moment before the update occurs, there are 144K megaflows in the cache. We see that the update causes about 104K deletions from the megaflow cache, followed by tens of thousands of upcalls. As a result, the throughput drops dramatically and takes a few seconds to recover.

This graph illustrates a general problem rooted in the megaflow algorithm. When inserting new rules that overlap existing ones with lower priorities, OVS must delete all megaflows that correspond to the existing rules (§3). While the magnitude of the throughput degradation depends on the rules being updated, the issue is significant in particular with high update rates.

Takeaway 4: modifying a handful of OpenFlow rules might significantly affect the throughput because of the increase in upcalls.



(a) OVS-CCACHE with NuevoMatch accelerating the megaflow cache.



(b) OVS-CFLOWS with NuevoMatch performing OpenFlow rule classification in the data-path.

Figure 6: Design options for integrating NuevoMatch with OVS. See §8 for the discussion why to choose one over the other.

4 Design Options and Challenges

Our analysis indicates two primary reasons for the OVS performance degradation: (a) poor scalability of the megaflow cache; (b) frequent upcalls to the control-path. In the following we consider two designs to solve these issues.

4.1 OVS with Computational Cache

To tackle the first issue, the most natural solution is to replace the megaflow cache with a more scalable NuevoMatch. This approach is appealing because it fits well in the existing OVS design. Here, NuevoMatch uses the megaflow cache as a remainder, and can be seen as an additional layer of caching for megaflows. We call this approach an *OVS with a computational cache* (OVS-CCACHE).

Figure 6a shows the proposed OVS-CCACHE design, depict-

Num. of OF rules	Upcalls per sec	Num. of Megaflops in cache	Training time est.	Coverage degrad. est.
1K	128	6.5K	30s	3.8%
100K	6.7K	102K	270s	7%
500K	6.4K	90K	250s	8%

Table 2: Characterization of rule update rate requirements in the megaflow cache. NuevoMatch training should be at least 100× faster to be applicable to the megaflow cache.

ing only the data-path. The control-path is unmodified. Incoming packets are first matched against the exact-match cache. A miss is then forwarded to the computational cache provided by NuevoMatch RQ-RMI models. The original megaflow cache serves the lookups which did not match in RQ-RMI. If missed again, the packet continues with the original OVS upcall mechanism.

When new megaflows are added to the data-path, they are first inserted into the original megaflow cache. The RQ-RMI model is periodically re-trained in a separate thread by pulling the added megaflows from the megaflow cache. When the training finishes, the old RQ-RMI models are replaced with the newly trained ones that already incorporate the new megaflows, and the megaflow cache is emptied.

Unfortunately, this solution inherits the performance limitations of the upcall mechanism, and thus would not scale well in case of frequent upcalls.

4.2 OVS with Computational Flows

To solve the issue of slow upcalls, one option is to apply NuevoMatch to the control-path classifier to speed up the handling of upcalls. Unfortunately, control-path tasks go well beyond OpenFlow rule matching, and it is unclear how to use NuevoMatch in this context. Specifically, the control-path effectively implements the algorithm for tracking and generating non-overlapping megaflows. This is the core of the control-path and it is tightly coupled with the rule matching logic. Thus, NuevoMatch is not suitable for control-path acceleration.

On the other hand, the excessive number of upcalls we observed stems primarily from the design choice to generate non-overlapping megaflows for the data-path. The fact that megaflows do not overlap is an essential feature in OVS design that allows fast-path performance optimizations, but it is also the one that deteriorates the throughput dramatically in case of frequent upcalls [3, 4].

Therefore, our proposed solution, *OVS with computational flows* (OVS-CFLOWS), leverages NuevoMatch to perform efficient packet classification directly on complex OpenFlow rules, without resorting to non-overlapping megaflows. As a result, we remove the megaflow cache mechanism and the

associated control-path logic, and obviate the need for upcalls. This approach, while more intrusive than OVS-CCACHE, holds the promise to boost OVS performance both with and without OpenFlow rule updates. How it fairs against OVS-CCACHE is one of the questions we answer in our evaluation.

Figure 6b shows the design of OVS-CFLOWS. While it resembles OVS-CCACHE, the difference is that NuevoMatch here is used to match OpenFlow rules instead of megaflows as in OVS-CCACHE. Similarly to OVS-CCACHE, updates are first inserted into the remainder (we use TupleMerge [6] for its implementation), and RQ-RMI models are periodically retrained to accommodate them.

4.3 Challenge: Slow NuevoMatch Updates

Unfortunately, in practice, NuevoMatch cannot support either OVS-CCACHE or OVS-CFLOWS. Recall that rule modification in the classifier requires retraining all its RQ-RMI models from scratch with the new, modified set of rules (§2.1). Therefore, the rule update rate is bounded by the training time of the models, which in turn depends on the number of rules in the classifier rather than on the number of modified rules.

In the following, we analyze the update rate requirements for OVS-CCACHE and OVS-CFLOWS, and show that NuevoMatch is over two orders of magnitude slower than required.

Megaflow cache rule churn. To understand the training rate requirements for NuevoMatch in OVS-CCACHE, we analyze the rule churn rate in the megaflow cache. For each OpenFlow rule size category we measure (1) the average rate of upcalls, which is equivalent to the rate of updates in the megaflow cache (we count insertions only, as NuevoMatch supports deletions without retraining), and (2) the average number of megaflows in the cache, which dictates the NuevoMatch training time if it were used to accelerate the megaflow cache.

Table 2 shows that for larger rule-sets (100K, 500K) there are about 6.5K upcalls per second, and the megaflow cache holds about 100K megaflows. Thus, NuevoMatch would have to retrain the model with 100K rules every 150 μ s. This is of course unrealistic: training a model of that size would require about 270 seconds according to the original paper.

The solution suggested by the authors of NuevoMatch is to accumulate the updates in the remainder and serve the queries from it while training. Thus, the *coverage* of the RQ-RMI model is lower during the training; hence, the performance is lower because more queries are served in the remainder. When the training is finished, the coverage improves, and a new round of training begins right away to catch up with the rules modified during the previous training round.

Unfortunately, this option is not practical either. If 6.7K rules get modified each second, the expected coverage degradation per second would be about 7% (see Table 2). If we accumulate the updates while training for 270 seconds, the coverage will become practically zero, nullifying the NuevoMatch performance benefits completely. For comparison, even to

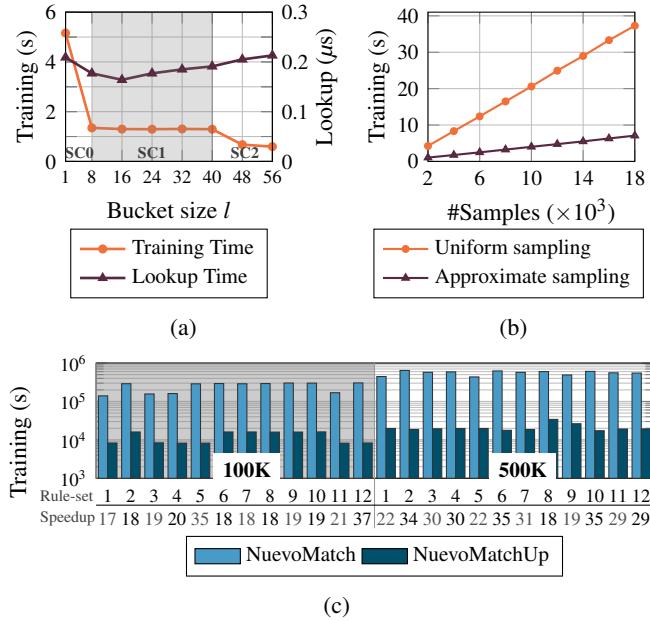


Figure 7: (a) The effect of the bucket size l on the RQ-RMI training and lookup times. See RQ-RMI size categories (SC) in Table 1. (b) Training using approximate sampling is faster. Here we train 500K rules using bucket size $l = 40$. (c) The training implementation of NuevoMatchUP is 20 times more efficient than that of NuevoMatch.

achieve the coverage of 25%, which is the cutoff suggested in the paper for NuevoMatch to provide minimum performance benefits, the training must complete within 4 seconds. This is almost *two orders of magnitude* faster compared to 270 seconds that NuevoMatch allows today. We use the same model as the original paper to produce these estimates: $E = R \cdot e^{-U/R}$, where R and U are the total number of rules and the number of updates respectively, and E is the expected number of rules in the model left after the updates.

OVS-CFLOWS update requirements. The update rate of OpenFlow rules varies between 400 to 338K updates per second [12, 13]. Supporting an average rate of 100K updates per second in NuevoMatch would require retraining every 500ms to achieve a coverage of 90% for a rule-set of 500K OpenFlow rules. Unfortunately, the actual NuevoMatch training time for a rule-set of that size is about 600s, which is over three orders of magnitude slower.

We conclude that *NuevoMatch training algorithm is too slow to support the update requirements of OVS in the considered designs.*

5 NuevoMatchUP: Speeding-up Updates

We introduce *NuevoMatchUP* (NMU), a series of enhancements to NuevoMatch which together significantly improve

its update rate by several orders of magnitude.

NMU introduces important changes to the RQ-RMI construction and training algorithms, as well as to their implementation. First, it enables creating much smaller (thus faster-to-train) RQ-RMI models by constructing iSets with overlapping rules. Second, it enables major improvement in training speed by cutting down the number of memory accesses. We now discuss these changes in detail.

5.1 Relaxing iSet Constraints

An iSet s is a set of rules associated with a field h_s for which rules do not overlap (§2.2). We relax the no-overlap constraint, by allowing overlap between a certain number of rules. Informally, a *relaxed iSet* is an iSet with up to l overlapping rules in field h_s , grouped in *buckets* (defined next).

We now describe the algorithm for constructing a relaxed iSet s on a header field h_s .

Lemma 1. *Given an OVS classification rule r and a packet header field h , the set of values in h that match r can be represented by an integer range, denoted as $h(r)$.*

The correctness of the lemma directly follows from the usage of prefix based wildcard representation in OVS.

Definition 1. *A bucket is a set of up to l rules. We say that two buckets b_1 and b_2 do not overlap with respect to the header field h if for any rule r_1, r_2 in b_1, b_2 respectively, $h(r_1)$ does not overlap $h(r_2)$.*

To create buckets that do not overlap with respect to the header field h_s , we sort the rules by their ranges in h_s , and iterate over them allowing up to l overlapping ranges per bucket. Whenever we encounter a rule with a range that does not overlap with its predecessors, we include it in a new bucket. If buckets contain less than l rules, we merge adjacent buckets while keeping the constraint to have at most l rules per bucket. Next, we use RQ-RMI models to *learn the distribution of the buckets* rather than the distribution of the rules [26].

Since the number of buckets is smaller by up to a factor of l than the number of rules, RQ-RMI models in NuevoMatchUP are smaller and train faster than in NuevoMatch (see Table 1). Of course, the cost of this optimization is a slower lookup: all the rules in the same bucket must be validated via a linear scan. This trade-off, however, turned out to be beneficial to accelerate training with a negligible slowdown for the lookup. Figure 7a demonstrates this trade-off using a representative rule-set (12-500K, see §7). Buckets of sizes $l = 8, 48$ change the RQ-RMI size category and dramatically improve the model’s training performance. Other bucket sizes ($l = 16, 24, 32, 40, 48, 56$) do not change the RQ-RMI size category and only add to the linear scan overhead.

5.2 Training via Approximate Sampling

In NuevoMatch, each neural net in RQ-RMI is trained using supervised learning on a labeled dataset S that is generated in advance. The dataset is sampled from an ordered set of ranges, R , sorted by the ranges' start values. An RQ-RMI model learns the function represented by the ordered set R : it maps an input to the index of the matching range. To learn this function, NuevoMatch samples from it uniformly [26]. This uniform sampling is expensive, as it requires to scan all the ranges and sample from them according to their relative sizes in the function input domain. This sampling must be done for each neural-network (NN) in the RQ-RMI model, sometimes multiple times to achieve the desired accuracy.

Our goal is to modify the sampling process to reduce the number of memory accesses from $O(|R|)$, which can be on the order of tens of thousands per NN, to $O(|S|)$, which is about several thousand per NN. Doing so is not trivial since the training converges faster when the samples are distributed with parameters $(\mu, \sigma) = (0, 1)$.

We make two observations. First, it is possible to analytically estimate the expectation μ and standard deviation σ of a uniform sampling of the NN input domain (see Appendix A.1), and thus enable correct normalization of the samples regardless of the way they are actually sampled. Second, given correct normalization, sampling R in a non-uniform way might only affect the model accuracy but not the lookup correctness, thanks to the search in the rule array (§2.2) that eliminates model approximation errors.

These observations allow us to accelerate the sampling process as follows. We generate a set of 32 samples per batch, each of the form (x, y) . First, we uniformly select a range r with index i from R . Second, we uniformly select a value $x' \in r$. We then generate a normalized $x = \frac{x' - \mu}{\sigma}$; $y = \frac{i}{|R|}$ as in the original algorithm.

In Figure 7b we train a model over a representative rule-set (12-500K, see §7) and get 4-5.3 \times faster training using approximate sampling.

5.3 Optimized Training Implementation

NuevoMatch uses a hybrid training approach that mixes Python code, TensorFlow, and a custom native library. In contrast, NuevoMatchUP is implemented in C++, which reduces its memory requirements, and takes advantage of the CPU SIMD instructions. Figure 7c shows a 23.8 \times geometrical mean speedup of NuevoMatchUP over NuevoMatch over all rule-sets. In this experiment we disable all algorithmic optimizations, highlighting the speedup due to the implementation.

5.4 Putting It All Together

Each of the described optimizations in isolation would not suffice to achieve the target performance goals to support the necessary update rate. However, when combined, they allow between two to three orders of magnitude faster training (depending on the rule-set), making NuevoMatchUP suitable for integration with OVS.

6 Implementation

We implement OVS-CCACHE in C as an additional OVS module, and NuevoMatchUP in C++ as an external library (*libnuevomatchup*). We add support for OVS-CFLOWS by changing existing components in several OVS modules¹.

Overview. OVS uses *poll mode driver* (PMD) threads for packet processing and *revalidator* threads for integrity. The flows² are kept in a dedicated *flow-table*, one per PMD thread, that supports a single writer and multiple concurrent readers. A PMD thread is responsible for inserting new flows into its flow-table, while the revalidator threads remove stale ones.

We modify OVS as follows. We introduce a single *trainer* thread to train all the NuevoMatchUP models used by each PMD thread. In addition, we add *manager* threads, one per PMD thread, for tracking the PMD flows, and create training tasks to accommodate the changes.

Concurrency. We use a fine grained locking with a spinlock per flow-table entry, and limit the number of occurrences in which we modify the flow-table. This mechanism is essential mostly for OVS-CCACHE, in which valid flows frequently migrate between the megaflow cache and the RQ-RMI models.

Training RQ-RMI models. At any given time, there are two instances of RQ-RMI models per *manager* thread: the one that is used by an active classifier in the packet processing pipeline, and the one being trained, referred to as a *shadow* model. In each iteration, a *manager* thread goes over all the flows, checks which are marked for deletion and which are new. Next, it enqueues the request with the modified rule-set to the trainer thread to retrain the shadow model. The rules added during training are updated in the remainder of the active classifier. When the training completes, the active classifier replaces its model with the newly trained shadow model, and the recently learned rules are removed from the remainder. This process repeats whenever the number of flows in the remainder is higher than 10%.

Data-path modifications. The megaflow cache constructs a new hash table whenever it encounters a previously unseen mask, and destroys it when it no longer holds flows. Since in OVS-CCACHE, megaflows frequently migrate between the megaflow cache and the RQ-RMI models, we enable the exis-

¹<https://github.com/acsl-technion/ovs-nuevomatchup>

²In this section we use the OVS terminology and refer to match-action rules of any kind, either megaflows or OpenFlow rules, as flows.

Name	Number of Packets	Unique 5-Tuples	Average Delay Between Packets (μ s)
CAIDA-short	100 M	6 M	1.68 ± 69.54
Mawi	237 M	15 M	3.39 ± 9.11
CAIDA-long	401 M	23 M	1.62 ± 119.20

Table 3: Evaluated traces.

tence of empty hash tables to reduce the number of hash table constructions and deletions to bare minimum.

6.1 Updates in OVS-CFLOWS

OVS-CFLOWS offers a new design trade-off for performing OpenFlow rule updates. Specifically, it allows trading the time it takes to activate the updated rules in the data-path for higher throughput during the update. When a network controller updates the rules, it might need to ensure that the updates are installed and visible to the data-path. In OVS and OVS-CCACHE, the acknowledgement to the controller is sent when the rules are installed in the *control-path*. The data-path pulls the rules on demand via upcalls.

In OVS-CFLOWS, we can implement two policies. The *instant update* policy updates the active classifier with the new rules immediately, pushing them into the remainder and thus applying them to the data-path without any delay. The *delayed update* policy first stores the new rules in a temporary structure not visible to the classifier, retrains the shadow model and only then updates the data-path.

As we will see in the evaluation, when a large number of updates is necessary, the instant update policy results in lower throughput while the new rules are being added due to reduced model coverage, but provides lower update latency from the perspective of the network controller. Delayed updates yield higher latency for the controller, but avoid the throughput degradation during the update. On the other hand, with only a handful of updated rules, the immediate update policy achieves low latency without affecting the throughput.

7 Evaluation

We perform end-to-end experiments and provide an in-depth analysis of the system performance using microbenchmarks.

7.1 Methodology

Setup. We use two machines connected back-to-back via Intel X540-AT2 10Gb Ethernet NICs with DPDK-compatible driver. All our tests stress the OVS logic thus the workload is CPU-bound and the network is not saturated.

The system-under-test machine (SUT) runs Ubuntu 18.04, Linux 5.4, OVS 2.13 with DPDK 19.11, on Intel Xeon Sliver

4116 CPU @ 2.1GHz with 32KB L1 cache, 1024KB L2 cache, and 16.5MB LLC. The load-generating machine (LGEN) runs a native DPDK application that generates packets on-the-fly according to a predefined policy, and records the responses from the SUT.

We configure both machines to use DPDK with four 1GB huge pages for maximum performance. We disable hyper-threading and set the CPU governor to maximum performance for stable results.

Synthetic OpenFlow rules. We generate OpenFlow rules using ClassBench [33], the standard benchmark for packet classification [6, 18, 19, 26, 35, 39]. ClassBench creates 5-tuple rule-sets that correspond to the distribution of three applications: Access Control List (ACL), Firewall (FW), and IP Chain (IPC). We generate rule-sets with 1K, 100K, and 500K rules, each size category with 12 rule-sets. We only generate rules for either TCP, UDP, or ICMP IP protocols. The mapping between the generated rule-sets’ names to their numbers appears in Appendix A.3.

Traffic traces. The traces are summarized in Table 3 and detailed below.

- (1) **CAIDA** [2]. The real trace from the Equinix data-center in Chicago, collected in January 2019. We use CAIDA-short in all experiments except for the one that needs longer trace (Figure 14) where we use CAIDA-long.
- (2) **MAWI** [37]. The real trace from a link between Japan and the USA, collected in April 2020.

Adjusting traces to rules. There are no published OpenFlow rules used for processing the packets in the recorded traces. We thus resort to the method used in prior work [26]. Specifically, we modify the packet headers in the trace to match the evaluated ClassBench rule-sets, as follows. For each unique 5-tuple we uniformly select a rule, and modify the packet header to match it. We also set all TCP packets to have a SYN flag. This method preserves the temporal locality of the original trace while consistently covering all the rules.

Packet generation policies. We use minimum-size 64-byte packets to stress the OVS classification logic. We evaluate the system with two load generation methods.

Constant TX rate. To ensure unbiased evaluation, we run the experiments with a constant-rate load generator, and report the highest rate that permits the average drop rate over the whole trace to be below 1%. The first 5% of the packets in each trace are used as a warmup and the associated drops are ignored. We do this as we observe that bootstrapping the megaflow-cache causes many packet drops. With 5% warmup packets, we achieve consistent throughput results.

Adaptive TX rate. We use the timestamps from CAIDA/MAWI packet traces but scale down the inter-packet delay to replay the packets at the highest rate that strives to maintain an average per-second packet drop rate below 1%. To achieve that, we dynamically adjust the

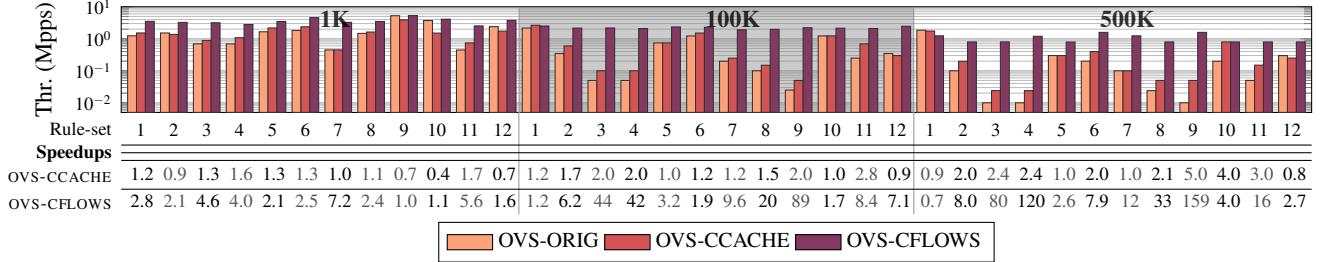


Figure 8: OVS-CFLOWS, OVS-CCACHE and OVS-ORIG on CAIDA-short using constant TX rate. Higher is better.

sending rate once per second: we cut it to half when the drop rate over the last second exceeds 1%, and increase by 50% otherwise. This methods provides a conservative estimate of expected system performance because of its simplistic congestion control which does not aggressively ramp up the throughput after drops.

Measurements. We measure end-to-end performance, i.e., receiving, processing, and sending packets back to the LGEN. We preload all OpenFlow rules into control-path.

OVS configuration. We use the default OVS configuration [22] both for the baseline and our designs: revalidator threads support up to 200K flows, flows with no traffic are removed after 10 seconds, and the *signature-match-cache* (SMC) is disabled. The EMC insertion probability is 20%. Connection tracking is not used. Unless stated otherwise, all experiments use a single NUMA node with one core dedicated to a PMD (poll mode driver) thread and another core dedicated to all other threads. Thus, the baseline OVS, ovs-CCACHE, and OVS-CFLOWS always use *the same number of CPU cores*.

NuevoMatchUP configuration. We use iSets with minimum 45% coverage, and train RQ-RMI neural nets with 4K samples. Similar to [26], we repeat the training until the RQ-RMI maximal error is lower than 128, and stop after 6 unsuccessful ones. We set $l = 40$, namely, each iSet bucket has at most 40 overlapping rules. We use the same RQ-RMI size categories as in Table 1. Due to the use of buckets, the largest size category is never used. We keep OVS’s flow matching mechanism that supports an arbitrary number of fields, but limit the iSet construction mechanism to use 5-tuples.

We train RQ-RMI models based on either all megaflows (for OVS-CCACHE) or OpenFlow rules (for OVS-CFLOWS). The model size is determined by the NuevoMatchUP algorithm to allow lowest error, fast training time and low memory footprint.

7.2 End-to-end Performance

Figure 8 shows the throughput comparison of OVS-CFLOWS, OVS-CCACHE and OVS-ORIG (unmodified OVS) for CAIDA-short with constant TX rate and without updates to the OpenFlow rule-set. The geometric mean speedups of OVS-CCACHE

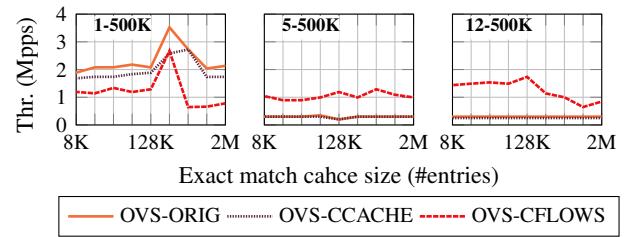


Figure 9: The effect of the exact-match cache size on throughput for the top three fastest rule-sets with 500K rules.

are $1.02\times$, $1.5\times$, and $1.9\times$ for 1K, 100K, and 500K OpenFlow rules respectively. Note that for OVS-CCACHE the computational cache is *constantly updated* with newly installed megaflows.

The same setup with OVS-CFLOWS yields higher speedups. OVS-CFLOWS is $2.6\times$, $8.5\times$, and $12.3\times$ faster than OVS-ORIG for 1K, 100K, and 500K OpenFlow rules, respectively. Not only is OVS-CFLOWS faster than OVS-CCACHE, but it also maintains a relatively stable absolute throughput for 100K and 500K rules. OVS-ORIG performance varies substantially across rule-sets of the same size, whereas OVS-CFLOWS shows more homogeneous behavior. OVS-ORIG has particularly low performance for larger rule-sets (e.g., 3,4 for 500K) due to a massive number of upcalls.

The performance trends with an adaptive TX rate are consistent with those obtained with the constant TX-rate (see Figure 18a in the Appendix). The speedups are still significant but more modest for two reasons: the adaptive TX fails to increase the sending rate fast enough after packet drops, which particularly affects the absolute throughput of faster OVS-CFLOWS. At the same time, it achieves higher average rate for lower-performing OVS-ORIG and OVS-CCACHE because it suffices to slowly increase the rate when the traffic pattern affords that. Rule-set 9-100K and 3-500K are the best illustrations of this effect.

Rule-set 1-500K performs differently from the rest. Here, OVS runs faster with 100K and 500K rules than with 1K rules. We find that this is due to the high temporal locality, which leads to a low upcall rate (over $3\times$ less than in other rule-sets for 500K) and a small megaflow cache. This analysis

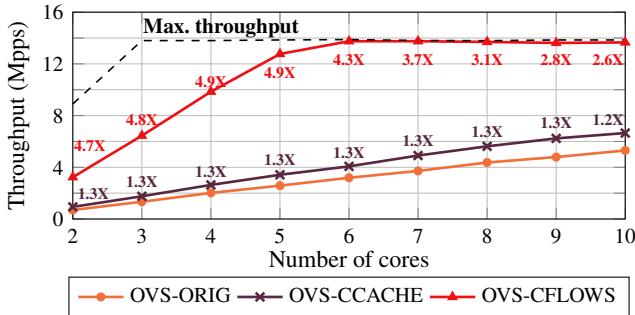


Figure 10: OVS throughput as a function of the number of cores. One core is dedicated to revalidator, manager, and trainer threads. For the rest, we allocate one PMD thread per core. The maximum throughput is measured with only EMC hits. The numbers refer to speedups vs. OVS-ORIG.

is corroborated by the experiments that vary the EMC size (Figure 9). This result motivates dynamic choice between the original and the suggested classification mechanisms as we discuss in §8.

The same experiments on the *Mawi* trace yield low throughput results for OVS-CCACHE and OVS-ORIG using constant TX rates due to the excessive number of drops. For the dynamic TX rate, the geometric mean speedups are: 2.1×, 18.2×, and 18.7× for 1K, 100K, and 500K rules for OVS-CFLOWS, and 1.02×, 1.4×, and 1.7× for 1K, 100K, and 500K rules for OVS-CCACHE.

7.3 Sensitivity to OVS parameters

The effect of the EMC size. We take the top three rule-sets with 500K rules that perform best for OVS-ORIG (rule-sets 1,5 and 12), and test their throughput with different Exact Match Cache (EMC) sizes (8K (default) to 2M), see Figure 9. The performance effect of the EMC size depends on the rule-set. The default size (8K) works reasonably well, whereas a too large EMC reduces throughput, likely because of the CPU cache contention. The relative performance of different designs, however, remains largely the same with the EMC of up to 128K entries.

Megaflow cache size. When the OVS megaflow cache reaches its maximum capacity it flushes all its contents. We validated that this never occurs in our experiments. Thus, the megaflow cache can practically grow as necessary, periodically evicting idle (for 10s) flows. This is the most favorable configuration.

Data-path scalability. We add PMD threads and pin them each to a separate core, while dedicating one more core for the revalidator, manager and trainer threads. We use the *CAIDA-short* trace with the constant TX setting, and report the results of a representative rule-set with 1K rules (3-1K) in Figure 10.

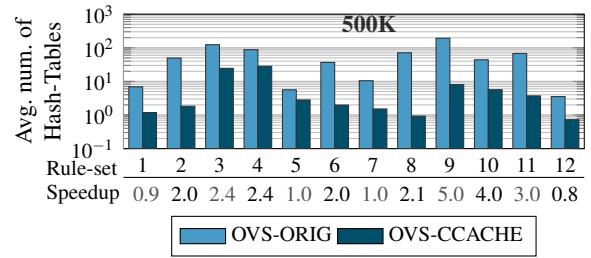


Figure 11: The average number of hash-tables in the megaflow cache on CAIDA-short trace. Lower is better. See full chart in the Appendix (Figure 18b).

This is the best-case scenario for OVS-ORIG because in larger rule-sets it is much slower. We measure the upper bound of the OVS forwarding performance by sending 100M packets that always hit the 8K flows-large EMC (black dashed line). For a 10Gb NIC, the performance saturates at 13.8Mpps, 93% of the line-rate³.

Figure 10 shows that OVS-CCACHE maintains a constant speedup of 1.3× over OVS-ORIG, even though more PMD threads lead to higher model retraining load. This is because the single trainer thread is fast enough to retrain models from eight PMD threads (nine cores in total on the graph). The additional, ninth PMD thread saturates the trainer. Without training fast enough, the scaling is no longer linear (1.2× speedup vs. 1.3× for fewer PMD cores). Thus, more PMD threads would require allocating additional trainer cores to maintain the speedup.

OVS-CFLOWS reaches the maximum throughput with five PMD cores (six cores overall), a 4.3× speedup over OVS-ORIG using the same number of cores. OVS-ORIG would have required about 26 cores (linear extrapolation of the current trend) to reach the same performance. Note that in contrast to OVS-CCACHE, models in OVS-CFLOWS are not retrained in the steady state between OpenFlow rule updates, thus the throughput scales linearly with more PMD threads without additional trainer cores.

7.4 Analysis of OVS-CCACHE

Understanding performance variability of OVS-CCACHE. Why does OVS-CCACHE is faster than OVS-ORIG for some rule-sets and is on-par or slower for others? The answer follows from Figure 11 which shows the average number of megaflow cache hash-tables traversed for OVS-ORIG and OVS-CCACHE. Recall that the classification is slower with higher number of hash-tables [3]. The computational cache achieves higher speedups when the number of hash-tables traversed by OVS-ORIG is large enough to justify inference computations instead of memory lookup. As a result, the performance

³ 14.88Mpps for 64B packets on a 10Gb NIC, considering bytes of Ethernet preamble and 9.6ns of inter-frame gap.

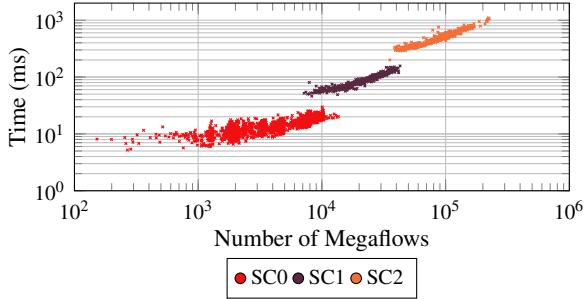


Figure 12: NuevoMatchUP training time in OVS-CCACHE as a function of number of megaflows and the model size category (Table 1). SC0=5, SC1=21, SC2=133 neural nets.

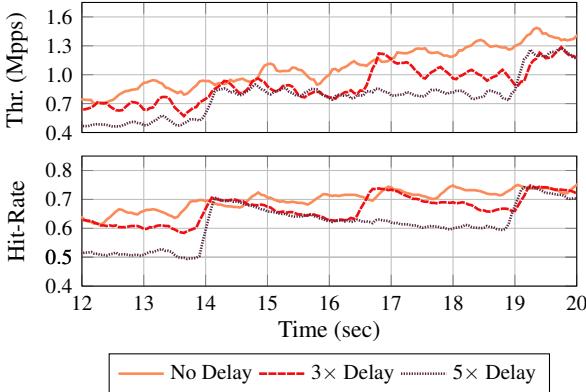


Figure 13: ovs-ccache throughput and computational cache hit-rate for different training rates, while adding megaflows. Rapid retraining is critical for high throughput.

savings from using NuevoMatchUP are higher in such cases.

Training in the data-path. In the following experiments, we generate packets at a constant rate of 5 Mpps. This setup saturates the OVS packet-processing pipeline and thus helps highlight the reasons why NuevoMatchUP improves the end-to-end performance.

We measure the actual training time for RQ-RMI models in the data-path during the experiment. To understand the training behavior, we measure the number of megaflows being used and the training time. We show the training time for each of the three used RQ-RMI size categories.

Figure 12 shows that the training time ranges from milliseconds for a small number of megaflows, to about one second for 200K megaflows. For comparison, NuevoMatch reported the training time of 270 seconds for a rule-set with 100K rules which NuevoMatchUP can train in 500ms - 540 \times faster.

Hit-rate and training time. We further analyze the dynamic throughput behavior of OVS-CCACHE when new megaflows are installed in it by the control path. We use a single rule-set with 100K OpenFlow rules (rule-set 9-100K), and vary the training rate while measuring the throughput.

Figure 13 shows that when new rules are just added the

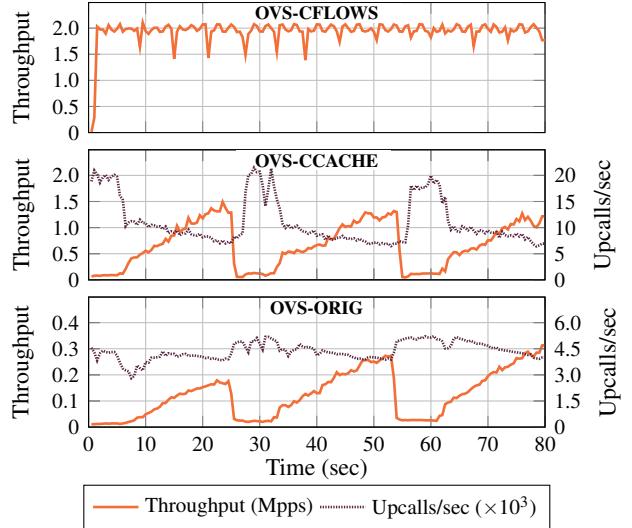


Figure 14: The throughput and number of upcalls over time.

throughput decreases initially, but then recovers. This behavior is expected. The rules are first installed in the original megaflow cache, which causes an increase in the number of hash-tables in it and the throughput drops. Also, the hit-rate in RQ-RMI models drops because the new rules are not yet part of the model. However, after the RQ-RMI model is retrained with the new rules, the hit-rate increases back, until the new rules get installed, and so on. Observe that the throughput is lower when the training is slower (i.e., 5 \times slower than the original rate) since in such cases the system cannot keep up with new rules. This experiment clearly demonstrates the importance of fast training provided by NuevoMatchUP.

Updates in OVS-CCACHE. We measure OVS-CCACHE average update rate for a different number of OpenFlow rules. We see 944, 11.6K and 11.2K updates per second, on average, for 1K, 100K and 500K rules, respectively.

Further inspection reveals that OVS-CCACHE sensitivity to upcalls affect its update rate, similar to the effect presented in Figures 4,5 for OVS-ORIG. Since we cannot explicitly control the upcalls, we test this by artificially delaying NuevoMatchUP updates and measuring the temporal behavior of the throughput, number of upcalls, and iSet coverage. We find that while NuevoMatchUP accelerates the megaflow cache, upcalls are still the dominating factor for its performance. See Appendix A.2 for details.

7.5 Analysis of OVS-CFLOWS

No upcalls in OVS-CFLOWS. We compare the throughput of OVS-ORIG, OVS-CCACHE and OVS-CFLOWS over time, sampled every 500ms. We use the CAIDA-long trace, so that each experiment is roughly 80 seconds long, and show the results of a single rule-set with 100K OpenFlow rules (rule-set 9-100K) while keeping the rules unmodified throughout the

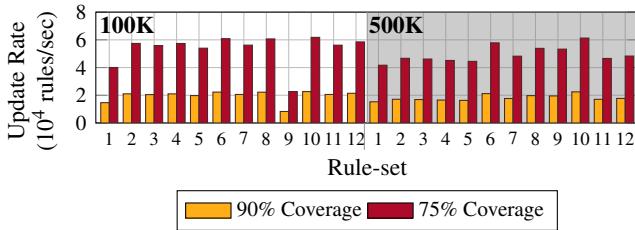


Figure 15: Max OpenFlow rule update rate of OVS-CFLOWS, for maintaining 75% and 90% NuevoMatch coverage.

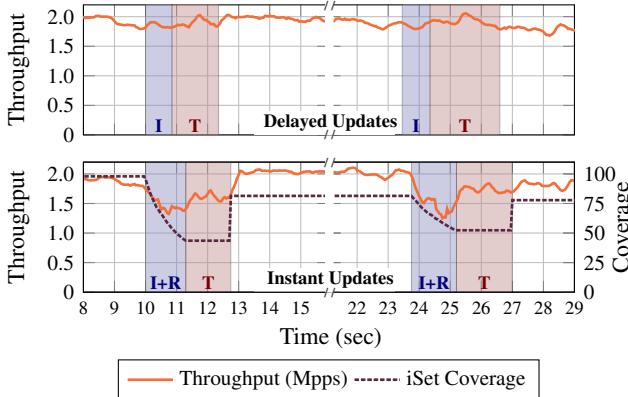


Figure 16: Update policies in ovs-cfows. I: iteration time, R: remainder time, T: training time.

experiments. Other rule-sets behave similarly.

The results in Figure 14 clearly illustrate the benefits of OVS-CFLOWS design (top graph). OVS-ORIG (bottom) and OVS-CCACHE (middle) suffer from significant performance fluctuations directly correlated with the number of upcalls into the control-path. This experiment corroborates our conclusions in Section §3. OVS-CCACHE inherits these performance problems because it simply replaces the megaflow cache with a faster alternative, but uses the same fast-/slow- path split. It does, however, improve the end-to-end throughput. The higher throughput of OVS-CCACHE is the reason why its upcall rate is proportionally higher than in OVS-ORIG.

OVS-CFLOWS avoids the use of upcall mechanism altogether, achieving consistently higher throughput and good scalability for a large number of OpenFlow rules.

OpenFlow updates in OVS-CFLOWS. We estimate the maximum OpenFlow rule update rate in OVS-CFLOWS for 100K and 500K rule-sets, as they pose the main challenge. Unfortunately, we could not measure the maximum update rate experimentally because of the slow OVS control-path that did not allow us to invoke updates back-to-back.

Our estimate of the update rate indicates the number of rules that can be updated per second in order to achieve 75% and 90% coverage by NuevoMatchUP. These are conservative

coverage values that were shown to result in small throughput degradation in NuevoMatch. To estimate, we measure the training time for each rule-set and compute the expected update rate according to the formula in §4.3. The results in Figure 15 show an average of 19K and 51K updates per second for 90% and 75% coverage, respectively. Both size categories achieve similar update rates since the average training time per rule is roughly the same, while the coverage deteriorates slower with more rules. These results assume the use of delayed updates which achieve higher throughput during the update.

Throughput during OpenFlow rule updates. We periodically add bundles of 125K new OpenFlow rules, so the number of rules increases throughout the experiment. We use this number of updates to make the dynamic system behavior over time more visible. We disable the EMC so that the measurements capture only NuevoMatchUP characteristics. We start the experiment with 100K OpenFlow rules, and measure the throughput and iSet coverage over time. We show the results on a representative rule-set (rule-set 9-500K), but the performance is representative of all rule-sets.

Figure 16 compares the delayed and instant update policies (§6.1). For the delayed policy, the time it takes for the data-path to receive the recent changes includes the time to process new rules (iterate over them) and to train, whereas in the instant updates setting, it includes the iteration and remainder update times. The training time depends only on the total number of rules, i.e., 225K and 350K in the first and second training sessions at 10 sec and 24 sec respectively. As expected, the instant update policy causes throughput degradation because the rules are added to the remainder, and thus the model coverage is low. Further, the accesses to the remainder data structure must be synchronized, creating contention. In this case, the use of delayed updates is beneficial as insertions do not cause measurable performance drop.

However, the instant update policy works well when the number of the inserted rules is small. An experiment using bundles of 100, 1K, and 10K new OpenFlow rules yields a 150ms-long drop in throughput with a maximum drop of 2%, 8% and 13% for 100, 1K and 10K rules, respectively. We start OVS with 500 OpenFlow rules and issue an update at $t = 10$ seconds. We use the CAIDA-short trace and the constant TX setting with 2.5Mpps. We use the same rule-set as in Figure 16 (rule-set 9-500K); other rule-sets behave similarly. We disable the EMC so the measurements capture only the characteristics of NuevoMatchUP. Figure 17 reports the throughput and iSet coverage within a three second time-frame surrounding the update.

8 Discussion and Future Work

Combining OVS-CCACHE and OVS-CFLOWS. Our evaluation shows that in most cases, OVS-CFLOWS is faster than both

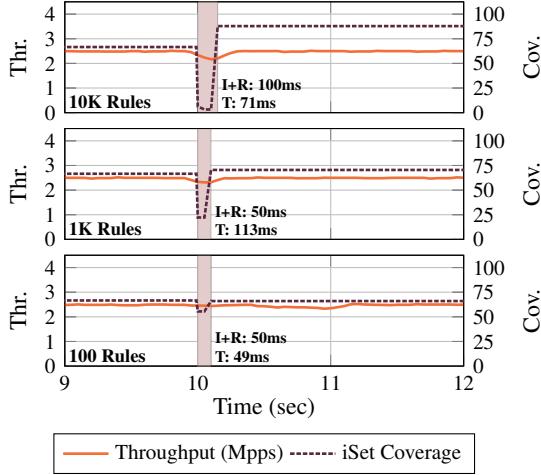


Figure 17: OVS-CFLOWS throughput and iSet coverage upon OpenFlow rule updates using the instant update policy. I: iteration time, R: remainder time, T: training time.

OVS-CCACHE and OVS-ORIG. However, there are cases where it would be desirable to switch between the classification mechanisms dynamically. The computational cache is beneficial when the number of hash-tables in the megaflow cache increases. This can be used to determine when to use it instead of the megaflow cache. Similarly, when the number of control-path upcalls increases, they become the main bottleneck, suggesting the use of OVS-CFLOWS.

NIC OVS offloads. OVS-CCACHE is compatible with the OVS ecosystem, and can be used with in-NIC OVS offloads [20]. In particular, it may accelerate the CPU handling of misses to the hardware OVS cache. Another question is how to use NIC OVS offloads with OVS-CFLOWS. It was shown that NICs become slow when the number of updates gets higher [13]. Thus, switching to OVS-CFLOWS whenever a high number of cache misses is detected may improve performance. We leave it for future work.

In-switch applications. Our work shows a practical use of NuevoMatchUP in packet classification. There are many similar tasks, e.g., longest-prefix matching in switches, which cannot scale due to small on-chip memory. We believe that NuevoMatchUP might help scaling up these tasks by compressing the indexing structure to save on-chip memory.

In-NIC NuevoMatchUP. RQ-RMI inference is a hardware-friendly task. Enabling its execution on the emerging data-parallel accelerators integrated with SmartNICs [21] may improve flexibility of the restricted packet classification of floating logic in NICs today.

P4 OVS. The possibility to use OVS with P4 in addition to OpenFlow was recently suggested [24]. Both the computational cache and computational flows are compatible with P4 as it uses the general structure of match-action tuples which is the fundamental building block for NuevoMatch.

9 Related Work

Packet classification. Software algorithms for packet classification are categorized into decision-tree approaches [9, 10, 18, 19, 28, 35, 39] and hash-table approaches [6, 22, 30]. NuevoMatch [26] is a new approach that shows superior performance for a larger number of rules, hence our choice to use it in this work.

OVS performance. Previous works have highlighted the problem of match-action fragmentation in OVS, and exploited it for mounting denial of service attacks on OVS [3, 4]. Ours is different: it analyses the causes of throughput degeneration and offers a solution.

Machine-learning in the data-path. Several works apply machine-learning models in performance-critical parts of the design, i.e., flash devices [11], RDMA key-value stores [36], programmable switches [38], and NICs [29]. To the best of our knowledge, ours is the first work that applies neural nets and integrates their training into a virtual network switch.

Trading memory accesses for computations. The pioneering work on learned indices [16] and several later works [5, 7, 14, 15, 17, 31] have shown the performance benefits of trading memory accesses for computations using machine-learning models, applying them to data-bases and key-value stores. NuevoMatch [26] extends these concepts and introduces the RQ-RMI data-structure that specializes in range-value queries. Our work improves the training technique of NuevoMatch by several orders of magnitude, making its integration with real-world systems feasible.

10 Conclusion

OVS is a leading virtual networking infrastructure used by many cloud systems. Our work demonstrates two designs which improve its throughput and scalability. We adopt a recent NuevoMatch algorithm for packet classification using neural nets, and integrate it with OVS. Our modifications to NuevoMatch make its use in OVS practical by accelerating its training by over three orders of magnitude. We show significant improvements in both steady-state throughput and update rate for large rule-sets on real-world packet traces. We believe that our work opens new opportunities to practical applications of neural-net based data structures in production networking systems.

11 Acknowledgements

We thank the anonymous reviewers of NSDI’22 and our shepherd Anuj Kalia for their helpful comments and feedback. This work was partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate. We gratefully acknowledge support from Israel Science Foundation (Grant 1027/18).

References

- [1] The OpenStack authors. The OpenStack project. <https://docs.openstack.org/liberty/networking-guide/scenario-classic-ovs.html>, 2021.
- [2] CAIDA. The CAIDA UCSD anonymized internet traces. http://www.caida.org/data/passive/passive_dataset.xml, 2019.
- [3] Levente Csikor, Dinil Mon Divakaran, Min Suk Kang, Attila Korösi, Balázs Sonkoly, Dávid Haja, Dimitrios P. Pezaros, Stefan Schmid, and Gábor Rétvári. Tuple space explosion: A denial-of-service attack against a software packet classifier. In *ACM CoNEXT*, 2019.
- [4] Levente Csikor, Vipul Ujawane, and Dinil Mon Divakaran. On the feasibility and enhancement of the tuple space explosion attack against Open vSwitch. *arXiv preprint arXiv:2011.09107*, 2020.
- [5] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpacı-Dusseau, and Remzi Arpacı-Dusseau. From Wisckey to Bourbon: A learned index for log-structured merge trees. In *USENIX OSDI*, 2020.
- [6] James Daly, Valerio Bruschi, Leonardo Linguaglossa, Salvatore Pontarelli, Dario Rossi, Jerome Tollet, Eric Torng, and Andrew Yourchenko. TupleMerge: Fast software packet processing for online packet classification. *IEEE/ACM Transactions on Networking (TON)*, 27(4):1417–1431, 2019.
- [7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. ALEX: An updatable adaptive learned index. In *ACM SIGMOD*, 2020.
- [8] The Linux Foundation. Kubernetes. <https://kubernetes.io/docs/concepts/services-networking/network-policies/>, 2021.
- [9] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *ACM SIGCOMM*, 1999.
- [10] Pankaj Gupta and Nick McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000.
- [11] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. Linnos: Predictability on unpredictable flash storage with a light neural network. In *USENIX OSDI*, 2020.
- [12] Danny Yuxing Huang, Ken Yocum, and Alex C. Snoeren. High-fidelity switch models for software-defined network emulation. In *ACM HotSDN*, 2013.
- [13] Georgios P. Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostic, and Gerald Q. Maguire Jr. What you need to know about (smart) network interface cards. In *PAM*, 2021.
- [14] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: A single-pass learned index. *arXiv preprint arXiv:2004.14541*, 2020.
- [15] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A learned database system. In *CIDR*, 2019.
- [16] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *ACM SIGMOD*, 2018.
- [17] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. A scalable learned index scheme in storage systems. *arXiv preprint arXiv:1905.06256*, 2019.
- [18] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. Cut-Split: A decision-tree combining cutting and splitting for scalable packet classification. In *IEEE INFOCOM*, 2018.
- [19] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification. In *ACM SIGCOMM*, 2019.
- [20] NVIDIA Networking (Mellanox). OVS offload using ASAP² direct. <https://docs.mellanox.com/pages/viewpage.action?pageId=39264792>, 2020.
- [21] NVIDIA. NVIDIA BlueField-2x AI-Powered DPU. <https://www.nvidia.com/en-us/networking/products/data-processing-unit/>, 2021.
- [22] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of Open vSwitch. In *USENIX NSDI*, 2015.
- [23] A Linux Foundation Collaborative Project. Open vSwitch. <https://www.openvswitch.org/>, 2020.
- [24] A Linux Foundation Collaborative Project. Open vSwitch and OVN 2020 fall conference. <https://www.openvswitch.org/support/ovscon2020/#D4>, 2021.
- [25] The DPDK Project. DPDK - data plane development kit. <https://www.dpdk.org>, 2020.

- [26] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. A computational approach to packet classification. In *ACM SIGCOMM*, 2020.
- [27] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. A computational approach to packet classification. *IEEE/ACM Transactions on Networking (TON)*, pages 1–15, 2021.
- [28] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *ACM SIGCOMM*, 2003.
- [29] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. Running neural networks on the NIC. *arXiv preprint arXiv:2009.02353*, 2020.
- [30] Venkatachary Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. In *ACM SIGCOMM*, 1999.
- [31] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. XIndex: A scalable learned index for multicore data storage. In *ACM PPoPP*, 2020.
- [32] David E Taylor. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys (CSUR)*, 37(3):238–275, 2005.
- [33] David E Taylor and Jonathan S Turner. ClassBench: A packet classification benchmark. *IEEE/ACM transactions on networking (TON)*, 15(3):499–511, 2007.
- [34] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open vSwitch dataplane ten years later. In *ACM SIGCOMM*, 2021.
- [35] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing packet classification for memory and throughput. In *ACM SIGCOMM*, 2010.
- [36] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered key-value store using remote learned cache. In *USENIX OSDI*, 2020.
- [37] WIDE MAWI WorkingGroup. Measurement and analysis on the wide internet (MAWI). <http://mawi.wide.ad.jp/mawi/>, 2020.
- [38] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning?: Toward in-network classification. In *ACM SIGCOMM HotNets Workshop*, 2019.
- [39] Sorrachai Yingcharonthawornchai, James Daly, Alex X Liu, and Eric Tornq. A sorted-partitioning approach to fast and scalable dynamic packet classification. *IEEE/ACM Transactions on Networking (TON)*, 26(4):1907–1920, 2018.

A Appendix

A.1 Approximate sampling

We show how to analytically calculate the expectation μ and standard deviation σ of a uniform sampling of an RQ-RMI neural-net input domain. We use the definitions and notations from [26].

RQ-RMI models contain several stages of *submodels* (neural-networks). In each stage, a single submodel is selected based on the output of the previous stage [26]. Let m be an RQ-RMI submodel.

The responsibility R_m of m is defined as the set of all values in \mathbb{R} that might reach m as inputs, formally $I_1 \cup \dots \cup I_n$, where $n \geq 1$ and $I_i = [a_i, b_i]$ are sorted non-overlapping intervals in \mathbb{R} .

For $1 \leq i \leq n$, define t_i as the sum of all weighted averages of I_j , $1 \leq j \leq i$. For ease of notation, $t_0 = 0$. Note that the intervals $[t_{i-1}, t_i] \subseteq [0, 1]$ do not overlap, and their location in $[0, 1]$ is relative to the weighted average of I_i .

For all $1 \leq i \leq n$, define the linear function $g_i(z) : [0, 1] \rightarrow R_m$ as follows:

$$g_i(z) = \frac{b_i - a_i}{t_i - t_{i-1}} \cdot (z - t_{i-1}) + a_i$$

In particular, $g_i(z)$ maps between the weighted average of I_i in $[0, 1]$ to $I_i = [a_i, b_i]$. The complete mapping between $[0, 1]$ to R_m can be described as the collection of all g_i functions, or as follows:

$$g(z) = \{g_i(z) | z \in [t_{i-1}, t_i], 1 \leq i \leq n\}$$

Given a uniform random variable $z \sim U[0, 1]$, the expectation μ and variance σ^2 of R_m can be described using $g(z)$:

$$\mu = \mathbb{E}[g(z)] \quad \sigma^2 = \mathbb{E}[g(z)^2] - \mathbb{E}[g(z)]^2$$

The two can be manually calculated from the equations above.

A.2 More on updates in ovs-ccache

We test the temporal behavior of OVS-CCACHE when facing upcalls and different update rates, similar to the analysis presented for OVS-ORIG (§3). Since we cannot control OVS-CCACHE update rate (§7), we artificially delay adjacent NuevoMatchUP training sessions. We use the same rule-set and trace as in Figure 4, and sample the system’s throughput, number of upcalls, and NuevoMatchUP iSet coverage, each 100ms.

The results shown in Figure 19 emphasize the importance of fast updates in OVS-CCACHE, as frequent upcalls cause the iSet coverage to drop to zero after just a few seconds, cutting the throughput by half ($t = 5$ sec). Note that the slow throughput of the system causes it to effectively digest the input at a lower rate, which in turn causes the upcall rate to go down as a result.

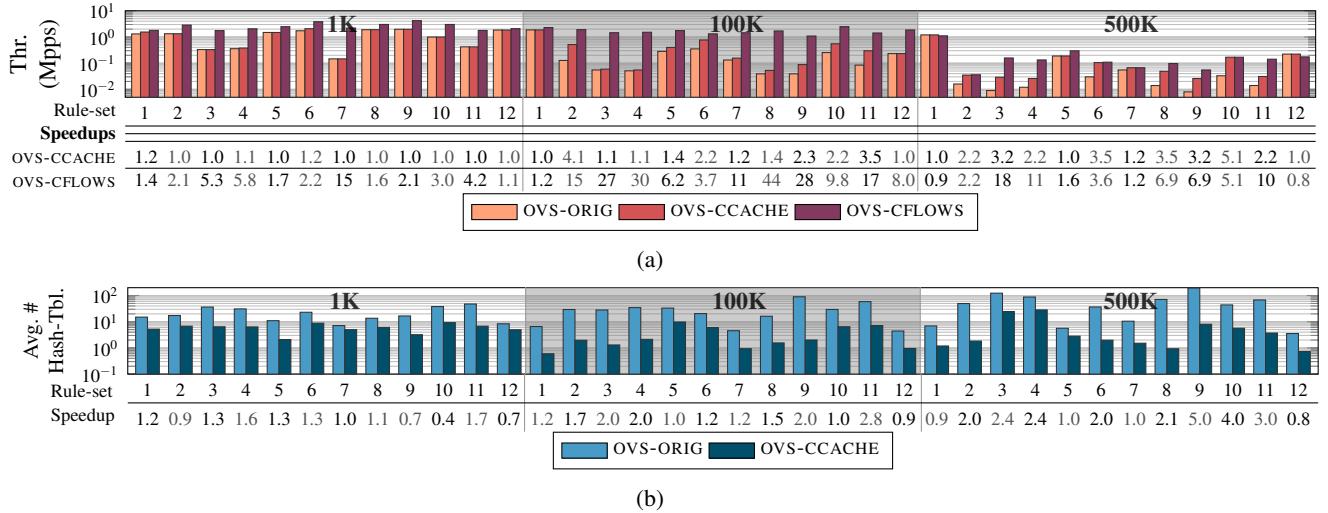


Figure 18: (a) OVS-CFLOWS, OVS-CCACHE and OVS-ORIG on CAIDA-short using adaptive TX rate. Higher is better. (b) The average number of hash-tables in the megaflow cache on CAIDA-short trace. Lower is better. This is an extended version of Figure 11.

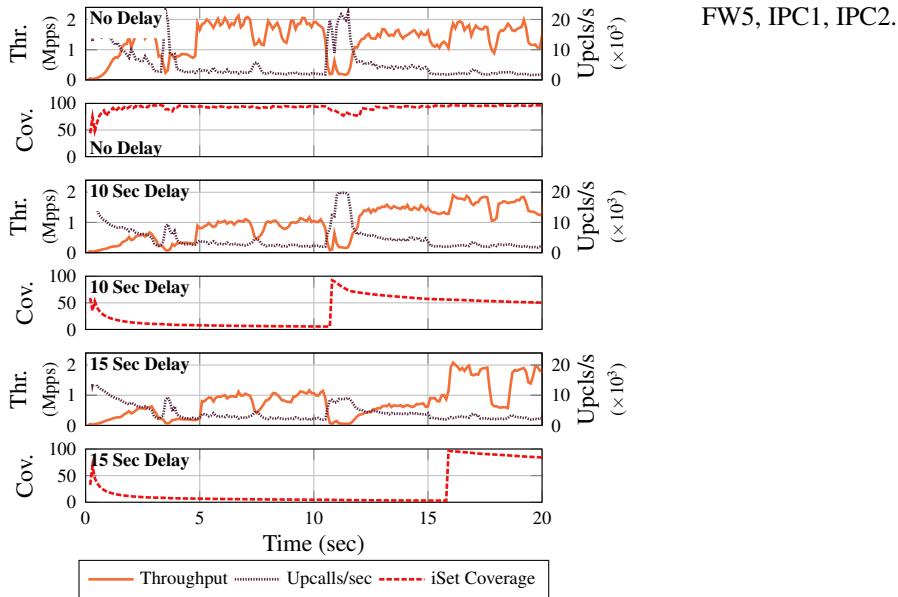


Figure 19: The effect of upcalls and NuevoMatchUP update rate on OVS-CCACHE throughput.

The results also show that upcalls still dominate the throughput as in OVS-ORIG ($t = 11$ sec), thus paving the motivation for OVS-CFLOWS.

A.3 Rule-set names

Rule-set names in Figures 7c, 8, 11, 15, 18a, and 18b by order:
ACL1, ACL2, ACL3, ACL4, ACL5, FW1, FW2, FW3, FW4,

Backdraft: a Lossless Virtual Switch that Prevents the Slow Receiver Problem

Alireza Sanaee[†], Farbod Shahinfar^{*}, Gianni Antichi[†], Brent E. Stephens[‡]

[†]*Queen Mary University of London*, ^{*}*Sharif University of Technology*, [‡]*University of Utah*

Abstract

Virtual switches, used for end-host networking, drop packets when the receiving application is not fast enough to consume them. This is called the slow receiver problem, and it is important because packet loss hurts tail communication latency and wastes CPU cycles, resulting in application-level performance degradation. Further, solving this problem is challenging because application throughput is highly variable over short timescales as it depends on workload, memory contention, and OS thread scheduling.

This paper presents Backdraft, a new *lossless virtual switch* that addresses the slow receiver problem by combining three new components: (1) Dynamic Per-Flow Queuing (DPFQ) to prevent HOL blocking and provide on-demand memory usage; (2) Doorbell queues to reduce CPU overheads; (3) A new overlay network to avoid congestion spreading. We implemented Backdraft on top of BESS and conducted experiments with real applications on a 100 Gbps cluster with both DCTCP and Homa, a state-of-the-art congestion control scheme. We show that an application with Backdraft can achieve up to 20x lower tail latency at the 99th percentile.

1 Introduction

Virtual switches (vswitches) play an important role in today’s data center networks operation [30, 33, 38, 68]. They are in charge of routing packets to one of the many competing microservices and applications running on a server that are communicating both locally and remotely [48, 66, 86]. They also provide isolation [61, 68, 87], enable load balancing [51], and perform packet encapsulation and decapsulation for secure virtual networking [30, 38, 39].

Virtual switches are fundamentally different from their physical counterpart. A physical switch has fixed port bandwidth, and its draining rate of output queues does not change over time. This is not the case for vswitches, as their draining rate of output queues depends on the ability of connected applications to consume packets. When packets arrive faster

than an application can process, queues inside the vswitch fill up and overflow, leading to packet loss. This is called the *slow receiver problem* [21, 44, 60, 73], and it hurts tail network communication latency and wastes CPU cycles, impacting application-level performance [22, 27, 91, 96].

In this paper, we show that slow receivers can manifest at short timescales and cause packet loss even in the presence of state-of-the-art congestion controls such as Homa [72] (§2.1). Moreover, CPU cycles are wasted in handling dropped packets, and this further increases latency and the already high software overheads of current network stacks [21, 72, 73], inflating the problem. Although there are existing approaches to mitigate packet loss (*i.e.*, bandwidth reservation [13, 49, 50], back-pressure [31, 43], credit-based hop-by-hop flow control [62], PicNIC [61]), they all have key limitations (§2.2). For example, because virtual ports bandwidth are variable over time, reservation schemes either lead to reduced network throughput or fail to prevent packet loss. Today’s backpressure flow control solutions suffer from severe Head-of-Line (HOL) blocking and congestion spreading, leading to reduced throughput across the entire cluster [44, 88, 99] and unacceptable latency for some applications [16, 65]. PicNIC [61, 79], a state-of-the-art solution to provide predictable performance in a multi-tenant data center, incurs high CPU utilization and consequent throughput degradation and HOL blocking for flows sharing a Virtual Machine (VM).

To prevent packet loss from the slow receiver problem, this paper presents Backdraft, a new lossless vswitch. Backdraft prevents packet loss while (1) avoiding HOL blocking, (2) reducing the required CPU cycles, and (3) preventing congestion spreading in the network core (§3). Our main insight is that, unlike physical switches, vswitches have abundant memory that can be used to support a large number of queues.

Leveraging this property, Backdraft assigns a separate queue for every single flow, preventing HOL blocking. To ensure that per-flow queuing is not prohibitive in its memory overheads, we introduce an approach that dynamically reclaims queues from idle flows and resizes them to accommodate in-flight packets from bursty flows.

Also, Backdraft uses separate queues for doorbells (notifications) and packet data to reduce the CPU overhead induced by per-flow queueing that can impact the vswitch performance. In this approach, the vswitch has only to poll the doorbell queue to find where the new to be processed data is located. By keeping the number of doorbell queues low, it is possible to greatly reduce CPU overheads, enabling per-flow data queueing and scaling to 100 Gbps switching performance.

Finally, Backdraft uses an overlay network between communicating vswitches. When a queue inside the vswitch begins to fill because of a slow receiver, Backdraft preemptively sends an Overlay Pause Frame (OPF) to the upstream vswitch responsible for the congestion with pause time and the slow receiver's bandwidth. This is practical because vswitches have a large amount of memory that can be used to store in-flight packets generated by the sender before receiving the OPF notification. Indeed, even buffering a full RTT of packets in a 100 Gbps network, a worst case of 1ms RTT would only require 12.5 MB of space (1 Bandwidth-Delay-Product - BDP), and end-hosts have GBytes of memory.

We implemented Backdraft on top of the BESS vswitch [3, 45] (§4), and evaluated it using a cluster of servers on CloudLab [75] equipped with 10 and 100 Gbps NICs (§6). We experimented with both standard and state-of-the-art congestion controls: in the first case we used unmodified POSIX applications leveraging the TAS TCP acceleration service [56]; In the second, we used Homa [72] with its DPDK implementation. When we ran a distributed application that performs RPCs, Backdraft in conjunction with Homa could lower its tail latency by up to 20x at the 99th percentile. With Memcached, instead, Backdraft could improve its goodput by up to 2.71x when compared to BESS. We also show that Backdraft does not suffer by HOL blocking and because of this can achieve 100 Gbps throughput in a cluster where a slow receiver is present. Finally, we demonstrate that Backdraft ensures high throughput with large number of queues. With 2K queues, throughput is 9x higher than BESS. This paper makes three contributions:

1. We make the case for building a lossless virtual switch by demonstrating the impact of slow receivers on packet loss and network performance using both DCTCP and Homa, a state-of-the-art congestion control algorithm.
2. We introduce Backdraft, a new lossless virtual switch that prevents the slow receiver problem and overcomes the drawbacks of state-of-the-art solutions: It (1) prevents packet loss, (2) removes HOL blocking, (3) increases throughput by eliminating wasted CPU cycles, and (4) avoids congestion spreading in the core network.
3. We implement and evaluate Backdraft on top of BESS using different congestion control mechanisms in a cluster of servers on CloudLab equipped with 10 Gbps and 100 Gbps NICs. We released our code under a flexible

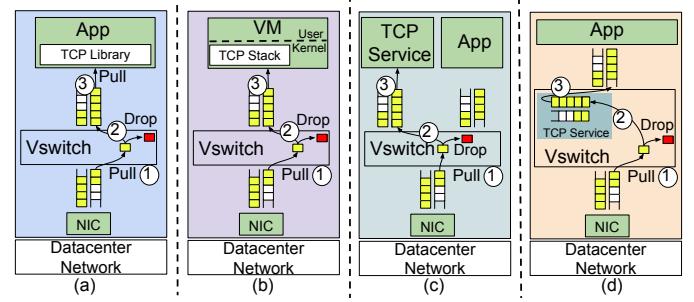


Figure 1: Various deployments of transport layer with respect to vswitches. (a) transport as a library. (b) transport as an OS service. (c) transport as a network function. (d) transport as a vswitch service.

open-source license to enable reproducibility¹.

2 Motivation

Virtual switches use shared memory queues to transmit and receive packets to and from connected end-points (Figure 1). Here, depending on the settings, the transport layer can be directly included into the application as a library (case a) [56], deployed in the kernel of a virtual machine (case b), used as a network service directly attached to the vswitch (case c) [59], or implemented in the vswitch (case d) [68]. Regardless, whenever the vswitch is ready to handle new data coming from the wire, it pulls a packet pointer from one of the NIC queues (point 1), performs processing and places it in the queue associated to the destination endpoint (point 2). Finally, the endpoint pulls the pointer and consumes the data (point 3). If this last step is not fast enough, the queue saturates and packets will be dropped at the vswitch. Notably, the discussed queue is not subjected to transport-level flow control mechanisms, so even if an endpoint has enough memory reserved for incoming packets (for example, TCP's receive window ensures there is space in the receive buffer), it is still possible for packets to arrive faster than the endpoint can process them and eventually get dropped. This issue has been acknowledged in the past, and it is called *the slow receiver problem* [21, 44].

2.1 The Slow Receiver Problem

There are many reasons for slow receivers, including allocation limitations [81], application-level limitations, load imbalance [19, 28, 32, 51, 52, 63, 71, 74], CPU performance variability [17, 25, 37, 42, 54, 66, 82, 97], and CPU/Memory contention [14, 35, 40, 41, 67].

To better understand this, we performed a number of tests on a 100 Gbps cluster (more information available in §6). First, we measured the achievable throughput of data-intensive (i.e., Nginx [8] and Memcached [7]) and network-only applications (iperf3 [6]) using an increasing number

¹<https://github.com/Lossless-Virtual-Switching/Backdraft>

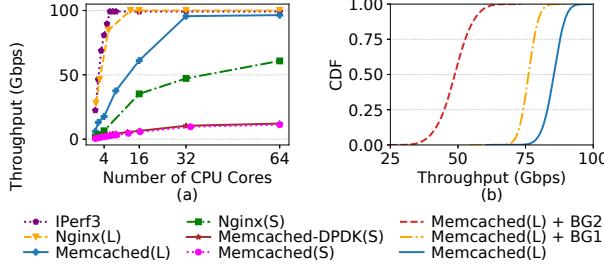


Figure 2: Maximum achieved throughput by Memcached, Nginx, and Iperf running with DPDK and Linux with large (L) and small (S) response sizes. (a) They require more than 6 cores to achieve 100 Gbps. (b) Memcached exhibits high throughput variability with and without the background workload. (BG1: on isolated cores. BG2: on shared cores)

of assigned processing cores. We also used different packet I/O frameworks (*e.g.*, standard Linux socket and DPDK) and different workloads. For Memcached, we used both small (200 B) and large values (4.8 KB) with sizes inspired by an analysis of caching at Twitter [92]. For Nginx, we served both small (4.8 KB) and large (1 MB) web pages.

Figure 2a shows the result of this experiment. We find that even iperf3, an application that only performs networking functionalities and no other specific processing, cannot hit 100 Gbps throughput with less than 6 cores. For other applications, even 64 cores might not be enough. Further, performance is highly dependent on the specific workload: Memcached using the Linux socket interface and serving 4.8 KB values with 32 cores achieves 16x higher bandwidth than the counterpart serving 200 B values. In contrast, Memcached can achieve 187 KRPS per core when serving 200 B items, while only 78 KRPS when serving 4.8 KB items.

Resource provisioning (OS scheduling) also plays a key role in application behavior [21, 73]. To better understand this, we run Memcached with 32 threads solely on bare-metal servers, where each thread resides on a separate logical core (the number of total logical cores is 64). Then, we use sysbench [58], which only exercises 32 logical cores, along with Memcached on the same machine. We evaluated both scenarios when Memcached and sysbench share CPU cores and when the two applications are isolated on different cores. Figure 2b shows that the Memcached server is unpredictable even *without* a background workload. When it is run with sysbench, its performance degrades by 12% or by 50% depending on the amount of contention. Moreover, the standard deviation of the throughput distribution increases by up to 1.71x.

Even worse, applications behavior can be highly variable and dependent on the workload [92]. We show this with experiments using Memcached and Nginx. To test the former, we used four clients generating a workload resembling the one experienced by Facebook [15]. For the latter, we used sixty single threaded clients requesting data from a copy of

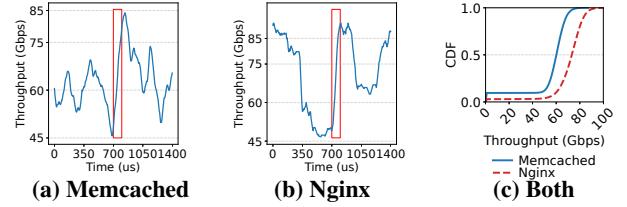


Figure 3: Throughput variability of Memcached (a) and Nginx (b) in a 1.4ms window. Throughput is highly variable over short timespans: *box* is 100 μ s. In the *box*, we can see over 40 Gbps variability in less than 100 μ s. (c) CDF of Memcached and Nginx throughput over the entire experiment.

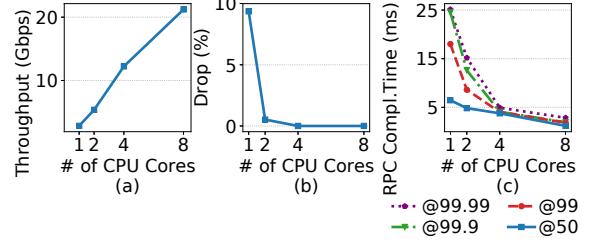


Figure 4: Throughput (a), packet loss (b) and RPC completion time (c) for a bidirectional RPC using Homa, the state-of-the-art transport protocol for data center networks. Packet loss can reach even 10% when only one core is assigned to the server application. RPC completion time can increase by \sim 9.3x at 99th.

the NSDI’21 website², a fairly light website composed of static pages. In Figures 3a and 3b, we show that performance variability in these applications is temporal. For instance, throughput varies about 45 Gbps in less than 100 μ s.

Furthermore, in Figure 3c, we illustrate the CDF of throughput for both Memcached and Nginx. Again, we can see variability: although they can both reach 100 Gbps, but for 50% of the time their throughput stays under 80 Gbps and 60 Gbps for Memcached and Nginx, respectively.

Observation I: Slow receivers are pervasive and can manifest at short timescales.

There are many new congestion control algorithms. However, even new algorithms still suffer from slow receivers. To show this, we ran a number of tests using Homa, a state-of-the-art transport protocol for data center networks [72]. Precisely, we performed a few tests where a client requests Remote Procedure Calls (RPCs) on a server, a dominating pattern in production data centers [57, 86], using a workload similar to the one experienced by Memcached servers at Facebook [72].

In Figures 4a and Figure 4b, we show that when the endpoint cannot process incoming packets fast enough, the drop rate increases. In this experiment, all packet loss occurs at the end-host, and the core network is loss free. This is particularly problematic because packet reception is expensive [69] and CPU cycles spent to eventually drop a packet are wasted resources that can amplify the problem. For example, it has been demonstrated that an increasing loss rate can cause ad-

²<https://www.usenix.org/conference/nsdi21>

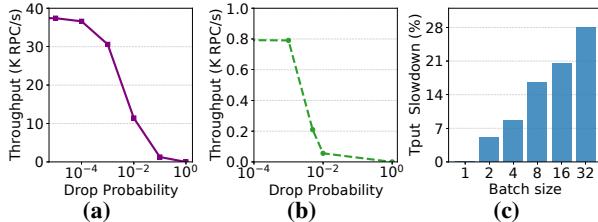


Figure 5: (a) The impact of packet loss on Homa’s and, (b) TCP’s throughput. Packet loss of 10^{-2} can halve the throughput. (c) The CPU cost of packet admission in PicNIC given different packet batch sizes. PicNIC’s out-of-order packet completion queues incur high CPU utilization.

ditional CPU cycles spent in handling the transport protocol, leading to fewer available cycles for data processing [21].

Further, Figure 4c shows that slow receivers lead to an increase in RPC completion time. This is because vswitch queues are shared across flows/RPCs belonging to the same application. As a queue becomes full, flows not responsible for the congestion (victim flows) will experience high latency. Specifically, we can see that even a slight slow down of the receiver application can cause the 99.9th percentile latency to hit values higher than 1ms. Those results show that even small amounts of packet loss can have a dramatic impact on the performance of receiver applications. We also performed a similar set of tests using DCTCP [12] to show DCTCP is also susceptible to high packet loss and report our results in Appendix (§ A.1.1).

To better understand the cost of packet loss, we performed an experiment where the vswitch is configured to drop packets according to a uniform probability distribution. We also used a standard TCP and Homa as transport protocols and measured the maximum sustainable throughput in terms of RPCs-per-second. In Figures 5a and 5b, we can see that even a *small percentages of packet loss* can dramatically impact performance. For example, the maximum sustainable RPCs-per-second can be halved with a packet loss probability of just 10^{-2} when using Homa. Also, the latency of Homa can reach milliseconds scale as packet loss exceeds 10^{-2} , as we show in Appendix (§A.1.2).

Observation II: Slow receivers cause sudden packet loss even in the presence of state-of-the-art congestion control mechanisms. Packet loss impacts network throughput and application service completion time.

2.2 Lossless Vswitching to the Rescue?

Packet loss at the vswitch is the source of many problems. However, there are already a variety of approaches that can be taken to avoid packet loss. These include reservations/rate-limiting, backpressure, credit-based flow control, or a combination thereof. Unfortunately, these have their own key limitations as discussed below and recap in Table 1.

Approach	Prevents packet loss	HOL blocking free	Avoids wasted CPU	Congestion spreading prevention
Rate-limiting [50]	X	X	X	X
Backpressure [31, 43]	✓	X	X	X
Credit-based [62]	✓	X	✓	X
PicNIC [61]	✓	X*	X	X
Backdraft	✓	✓	✓	✓

Table 1: A comparison of existing approaches to reducing packet loss. (*) PicNIC only prevents HOL blocking for flows coming from different VMs.

Reservation Schemes (Rate limiting). One option could be to rate-limit traffic according to bandwidth reservation schemes [13, 49, 50]. Although this is a good option for physical switches, it is not applicable in the virtual context. This is because such schemes assume that the line-rate processing is known in advance and deterministic. While this is the case for hardware switches, it is not for virtual ones.

Backpressure. Another option is to use a backpressure flow control scheme such as PFC [31] or BFC [43]. The main idea here is to send a pause message to the upstream switch before incurring a buffer overflow. Unfortunately, both PFC and BFC have key limitations that prevent them to be used as viable solution in a vswitch. The former might cause HOL blocking [29] and congestion spreading [44, 99] when the PAUSE frame from the vswitch reaches the upstream hardware switch. The second relies on the observation that most flows in a data center network are relatively short at today’s 100 Gbps line-rates to avoid HOL blocking from priority hash collisions inside the network core. However, this assumption breaks if slow applications connected to a vswitch are allowed to generate PAUSE messages. In this case, slow receivers will cause congestion spreading, and hash collisions will result in reduced throughput of victim flows from line-rate (100 Gbps) to the rate of the slow receiver.

Credit-based Flow Control. Hop-by-hop credit-based flow control is another mechanism for ensuring zero packet drop [62]. Unfortunately, this technique requires an RTT to request credits and specific support from switches which makes it difficult to be deployed on production networks [24]. Similar to backpressure schemes, credit-based flow control requires packets to be buffered at switches when there are no credits available, leading to HOL blocking.

Observation III: Standard lossless techniques either cannot be used in a virtual context or cause severe HOL blocking and congestion spreading.

Other Approaches (PicNIC). PicNIC [61, 79] is a state-of-the-art solution to provide predictable performance in a multi-tenant data center where per-VM service level objectives (SLO) must be met. PicNIC takes an end-to-end approach to provide backpressure from receivers to senders and aims at preventing HOL blocking at the transmit-side by introducing a packet admission control system where descriptors may be completed out-of-order. This is implemented using a specific

Backdraft Component	Purpose	Expected result
Dynamic per-flow queuing	Avoids HOL blocking, On-demand memory usage	Mitigates tail latency, Improves throughput, Flexible packet scheduling, Prevents pause frame flood.
Doorbell queue	Avoids wasted CPU	Avoids extra pause frame generation, Saves network bandwidth, Alleviates the slow receiver problem.
Virtual switch backpressure overlay network	Avoids packet loss, Vswitch-level flow control, PFC/BFC compatibility	Avoids extra pause frame generation, Saves network bandwidth, Alleviates the slow receiver problem.

Table 2: Backdraft’s components and their contributions

feature available in virtio interface [10, 78]. To understand its associated cost, we conducted an experiment where two end-points are connected to a vswitch on the same host. Each end-point is assigned a single core. Then we experimented with both *out-of-order* and *in-order* completion queues in the vswitch. Figure 5c, depicts that the out-of-order packet completion approach is slower than in-order by 20% and 28% when using a batch size of 16 [68] and 32 [3], respectively. Further, this is a baseline with only one core, and these overheads increase with a larger number of cores and queues. Thus, irrespective of application behavior, PicNIC imposes a high toll on performance. Furthermore, while PicNIC can successfully provide predictable performance for flows generated by different VMs, it does not have any mechanisms to ensure isolation between flows coming from the same VM as the out-of-order completion queues have a per-VM granularity, meaning that the slow-receiver problem can still happen and affect all the flows within the same VM.

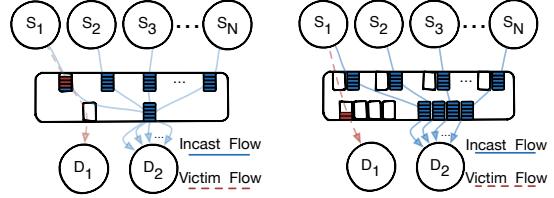
Observation IV: PicNIC can only isolate slow receivers at a per-VM granularity. It also imposes high CPU utilization and causes throughput degradation.

3 Backdraft Overview

Backdraft is a vswitch that provides lossless networking with higher throughput and lower CPU overheads than lossy switching, and Backdraft does not suffer from HOL blocking or congestion spreading. Backdraft achieves its goals by using three main components: (1) Dynamic Per-flow Queuing (DPFQ); (2) Separate queues for doorbells and data; and a (3) Virtual switch overlay network used for backpressure. Table 2 summarizes the purpose and effect of each component.

Dynamic Per-Flow Queuing (DPFQ): To avoid HOL blocking, Backdraft assigns a separate queue for every single flow in the vswitch, where a flow is an individual TCP connection. However, preallocating queues and memory for the worst case number of flows and burst sizes would be prohibitive. To ensure that per-flow queuing is not prohibitive in its memory overheads, we introduce a new approach that dynamically reclaims queues from idle flows and dynamically resizes queues to accommodate in-flight packets from bursty flows (DPFQ).

By enabling per-flow queueing, Backdraft fundamentally eliminates the HOL blocking caused by slow receivers and



(a) A traffic pattern where using backpressure buffers from HOL blocking. **(b) An illustration of why using separate queues for each virtual switch port avoids HOL blocking.**

Figure 6: Queuing and HOL blocking with backpressure.

incasts. HOL blocking only occurs when flows share a queue, and every flow in DPFQ is served by its own queue (Figure 6b versus Figure 6a). DPFQ is possible because end-host memory is not as limited as in physical switches [43, 84]. However, the challenge is ensuring that DPFQ does not incur prohibitive memory overheads even though the number of active flows is potentially large [77]. Over-provisioning leads to memory pressure, while under-provisioning forces flows to fall back to sharing the same queue, potentially incurring HOL blocking.

To solve this issue, Backdraft introduces a new approach to efficiently resize queues on demand. Although all memory for queues and packet buffers is allocated when the process is created to avoid performance stalls, queues are dynamically allocated and reclaimed from flows as they start and stop, and queues are dynamically grown by combining queues as needed to accommodate bursts of packets. This dynamism allows for efficient per-flow queuing without increasing memory overheads. Our insight is that the total amount of congestion that can occur in a vswitch is limited by things like the line-rate of the NIC and not by the number of active flows. Given the same amount of memory, DPFQ enables the same congestion tolerance as a single queue.

DPFQ introduces a new interface to the vswitch. However, it is still possible to support DPFQ without modifying applications. For example, most TCP applications (e.g., POSIX sockets applications) already perform per-flow operations. In this case, only the TCP stack needs to be modified to support DPFQ. Further, Backdraft supports legacy DPDK [47] and Netmap [76] applications that expect a shared queue interface with a vswitch by performing DPFQ inside the vswitch.

Doorbell Queues: The CPU overheads of a vswitch increase linearly with the number of queues that need to be polled [41], and data center workloads may have thousands of flows [18, 77]. Backdraft overcomes this limitation by using separate queues for data and doorbells. For each endpoint, there is a data queue for each flow and a doorbell queue for each core. To send data, an endpoint first enqueues packets in data queues then sends a doorbell message to the doorbell queue. This allows the vswitch to poll only an application’s doorbell queue to learn about new data.

Doorbell queues also provide a mechanism for applications to communicate scheduling information about the rel-

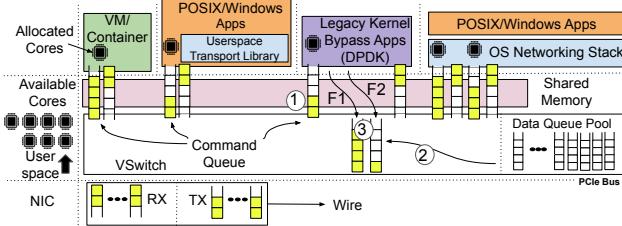


Figure 7: An overview of Backdraft’s architecture.

active priorities and weights of all active flows. Similar to prior work [79, 80, 87], this enables Backdraft to perform programmable scheduling and ensure that the appropriate queues are scheduled first to ensure low latency.

Virtual Switch Backpressure Overlay Network: When combined with backpressure, DPFQ can avoid both packet loss and HOL blocking for traffic local to the vswitch (server). However, if Backdraft runs out of buffer space and data is still incoming from a NIC, it must send a pause frame to the upstream TOR switch connected to the NIC to avoid packet loss when interfacing with a lossless network core, and it must drop packets when interfacing with a lossy network core. Unfortunately, generating pause frames can lead to congestion spreading, while dropping packets has a significant impact on network performance (Figure 5a and Figure 5b).

To avoid such problems, Backdraft builds an overlay network out of vswitches where Backdraft eagerly sends pause messages on the overlay network to the upstream vswitches that are causing congestion and either lazily sends pause messages to the upstream physical switch or lazily drops packets. This enables the local congested vswitch to continue buffering packets while waiting for the remote vswitch to react without causing congestion spreading. Additionally, DPFQ ensures that there is no congestion spreading inside the upstream vswitches because it is possible to pause only the flows responsible for the congestion.

With a lossless network core, the difference between the overlay pause threshold (Th_{over}) and the network pause threshold used for PFC or BFC (Th_{net}) determines the amount of data that can be buffered while waiting for the upstream vswitch to react. If the difference in bytes between these two thresholds is greater than the current network’s bandwidth delay product (BDP), *i.e.*, the RTT times the network line rate ($Th_{over} - Th_{net} > RTT * BW$), then it is possible for the overlay network to react to a slow receiver without needing to send a network-level pause message. Because buffering 1ms of packets at 100 Gbps line-rate only requires 12.5 MB of buffering, it is easy to buffer multiple BDPS of packets in a vswitch with low overheads.

4 Design

Applications connect to Backdraft through queues implemented on top of shared memory, and both applications and

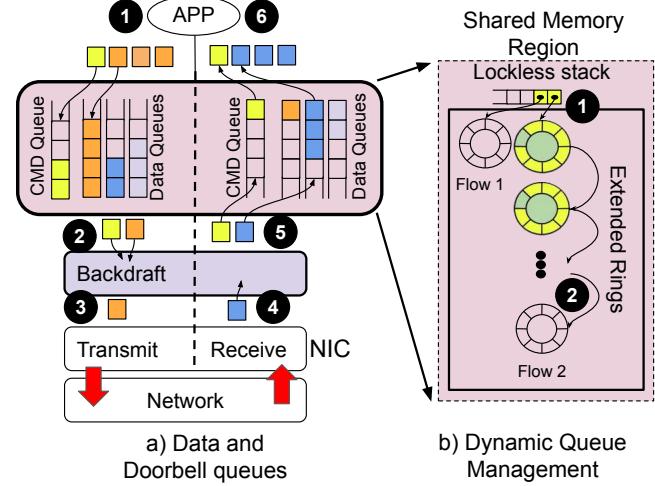


Figure 8: (a) Life cycle of control messages and data messages.
(b) Data queue pool memory overview in DPFQ.

the vswitch detect packets by polling. Native Backdraft applications use doorbell queues and data queues in both RX and TX directions. However, Backdraft also supports legacy DPDK applications that only use data queues to send packets as well as standard applications using the kernel networking stack through a custom kernel driver. Currently, Backdraft is designed to be a userspace vswitch although its key ideas are also applicable to kernel-space switching.

Figure 7 provides an overview of Backdraft. First an application sends control messages to the vswitch ①. Upon the arrival of the doorbell message, Backdraft allocates the appropriate data queues ②. Finally, data packets exchange starts ③. Similarly, as new flows start and stop, an application can send more doorbell messages to allocate or release additional data queues. Backdraft supports dynamic queue allocation and resizing via a linked list structure to efficiently manage packet buffers. The rest of this section discusses the design of the Backdraft components in more detail.

4.1 Doorbell Queues

Backdraft uses doorbell queues to reduce the CPU overheads of DPFQ and achieve high throughput. DPFQ increases the number of available queues, and polling them all is inefficient. Checking for outstanding packets costs a memory access, which requires ~ 100 cycles per queue. There are two ways doorbell queues reduce polling overheads: First, only doorbell queues and not data queues need to be polled. Second, the total number of doorbell queues is kept small. To support parallelism, an application needs at most one hardware thread per doorbell queue.

Figure 8a illustrates the control flow between doorbell and data queues. ① The application generates a doorbell message, notifying the vswitch. ② The vswitch receives outstanding packets. If the destination is a remote server, ③ the vswitch sends the packets to the NIC, and then ④ the packets arrive

at the destination vswitch. Once the packet is at the receiving vswitch, ⑤ the vswitch places the received packet in the appropriate per-flow queue and then generates doorbell message for the application. Finally, ⑥ the application receives a doorbell message and then polls the data. Additionally, all of the command messages in a command queue are read at once in a batch to ensure there is no HOL blocking.

4.2 Dynamic Per-Flow Queuing (DPFQ)

It is important to ensure that DPFQ does not put pressure on memory; hence, DPFQ dynamically reclaims, reassigned, and resizes queues to reduce the memory pressure.

When an application initially connects to Backdraft, the data queue descriptors are negotiated between the vswitch and the application. As applications push packets to buffers, Backdraft allocates individual queues on demand (Figure 8b). To prevent applications' address spaces from being exposed to others, separate shared memory regions and pools of queues are used for each application. This separation of buffering across applications ensures isolation.

Backdraft dynamically resizes queues to absorb packet bursts while minimizing memory overheads. To this end, Backdraft allocates ring buffers of fixed size and then links them together to form and extend queues (Figure 8b-2). Before a ring buffer gets full, Backdraft extends the queue by placing a pointer to a new ring buffer instead of a packet buffer in the overloaded queue. This enables it to learn about an extended queue without any race conditions. Then, once a flow becomes idle, Backdraft reclaims the initial queue into a pool that it can allocate to other queues.

Backdraft pre-allocates all queues at boot time and pushes all the pointers to these queues in a lockless stack. The number of pre-allocated queues can be configured depending on the workload but we used 50 queues for the experiments of this paper. Backdraft benefits from the lockless stack in two ways: First, this structure improves cache efficiency as a pushed pointer can be used immediately from the top of the stack. Second, Backdraft is capable of supporting multiple threads accessing the data queue pool. When a new flow arrives at Backdraft, it borrows a pointer to a queue from the stack and enqueues packet pointers in the queue (Figure 8b-1). If this queue becomes fully occupied, Backdraft borrows another pointer and links it to the previous one as is depicted in Figure 8b-2.

Backdraft does not deallocate empty queues, nor does it leave empty queues allocated to idle flows. Instead, it reclaims empty queues and pushes them back to the lockless stack. This helps Backdraft to reuse reclaimed queues promptly without deallocating them. Backdraft is only responsible for queue assignment/reclamation leading to no race conditions. An entire queue can be reclaimed once there are no outstanding packets in the queue. Full reclamation only happens when a receiver application notifies Backdraft by means of a doorbell

message about the emptiness of a data queue. Similarly, for new queues, applications must send a doorbell message to Backdraft requesting a queue corresponding to the new flow.

Both RX queues and TX queues can be extended. RX queues are frequently extended to tolerate bursts. In contrast, TX queues are only extended for flows with large BDPS, and there is no need to extend TX queues beyond a BDP in length. Instead of extending transmit queues beyond a BDP in length, an application can infer that a transmit queue being full is because of congestion or a slow receiver, and DPFQ enables applications to react to congestion. Many applications can simply keep packets buffered inside a TCP stack until the queue drains. However, it is also possible for some applications to mutate or even discard packets to reduce load.

Legacy Interfaces: Backdraft is backward compatible with both POSIX applications and DPDK applications. For the former, there are two ways to interface with Backdraft: (1) Backdraft uses a userspace TCP library that dynamically links to legacy socket applications (TAS [56]). (2) Packets can be received from the kernel through a custom networking driver. This is useful for applications that require features not yet supported by our library, *e.g.*, PF_RING.

4.3 VSwitch Backpressure Overlay Network

When there is congestion because of a slow receiver, Backdraft uses backpressure and sends pauses messages to the upstream sources of traffic to avoid packet losses. However, Backdraft is unique in that there are two different types of pause messages that it can generate: Overlay Pause Frames (OPFs) that are sent on a vswitch-to-vswitch overlay network and network-level pause frames that are sent hop-by-hop across the physical topology by a backpressure flow control scheme like PFC or BFC [43]. Backdraft implements PAUSES internally by function calls instead of sending PAUSE frames throughout the pipeline because this reduces CPU overheads. PAUSE frames are only created if the PAUSE frame is destined for a remote end-point, which enables Backdraft to provide lossless forwarding across a cluster.

To avoid congestion spreading, Backdraft eagerly generates OPFs. When the occupancy of a receive queue crosses a configurable threshold ($T_{h_{over}}$), Backdraft generates an OPF and sends it to the upstream Backdraft virtual switch that is causing congestion. Because there is only one flow per receive queue in Backdraft, only one message needs to be generated.

OPFs contain three pieces of information that are used by the upstream vswitch: 1) flow identifier, 2) pause time, and 3) new transmission rate. When an upstream vswitch receives an OPF, it pauses the input queue for the specified pause time, and then it applies a transmission rate-limit on the input queue.

Although prior backpressure schemes only send a pause time, sending a rate in an OPF is important to avoid persistent on/off congestion bursts from transmitters restarting after

being paused. To support this, Backdraft tracks an estimated receive rate (R_{recv}) using an exponential weighted moving average (EWMA) for each receive queue as it delivers packets, and it uses this rate when generating an OPF. The pause time is set as $(Th_{curr} - Th_{goal})/R_{recv}$ where Th_{curr} is the current length of the queue and Th_{goal} is the target queue length, which is equal to the batch size of packets read by the TCP stack by default to help ensure efficient CPU utilization.

The biggest concerns with respect to choosing values for Th_{goal} and Th_{over} are in avoiding starvation and reducing CPU overheads. Starvation is possible if the receiver vswitch either underestimates the end-point's rate or sets too large of a pause value. Th_{goal} provides headroom to avoid this, and if starvation is observed to be a problem with a running application, both the application and the vswitch can increase this value. In contrast, to avoid congestion spreading, it is desirable to set as large of a value for Th_{net} as possible because Backdraft generates PFC/BFC messages that will be processed by the upstream switch when this threshold is exceeded. This value can be as large as the maximum length of the queue minus the 1-hop bandwidth-delay product between the server and its TOR switch ($1\text{-hop RTT} \times \text{line-rate}$).

On the whole, sending OPF messages significantly reduces CPU utilization by preventing packet drops. However, to reduce the CPU overheads of OPF messages, Th_{over} is set to be at least one batch size of packets larger than Th_{goal} to not interfere with batching. Further, to avoid excessive OPF generation, Backdraft generates a new OPF message only if the previous OPF message pause time has gone past. When the pause time passes, Backdraft checks the queue length to decide whether to generate another OPF message or not.

5 Implementation

Backdraft builds upon the BESS virtual switch [3] (commit [0145a1c](#)). We have extended the TAS TCP stack [56] (commit [a1c158f](#)) to support TCP legacy applications. Further, we have implemented a Homa open-loop app based on the Homa DPDK library (commit [392b577](#)) and altered the DPDK driver to interface with Backdraft. Our changes to BESS amount to about 3.5K LOC, and our changes to TAS and Homa required about 100 and 500 LOC, respectively. Apps running TAS and Homa both connect to BESS via a DPDK vHost user port.

6 Evaluation

In this section, we evaluate the performance of Backdraft and demonstrate that Backdraft is able to prevent packet loss while providing 100 Gbps switching capabilities and without incurring in HOL blocking.

Experimental cluster: We used two different types of clusters from CloudLab [75]. On the first, we were able to use PFC to perform experiments with a lossless fabric. This clus-

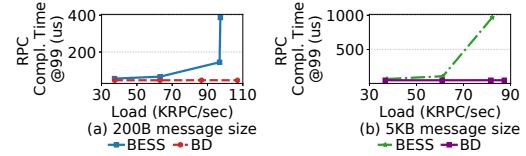


Figure 9: Performance of a victim RPC with Homa in the presence of an increasing load generated by a competing application. We used two different message sizes and either BESS or Backdraft as vswitch. The victim RPC is less impacted by the competing workload in the presence of Backdraft.

ter has 6 servers, and each server has an Intel Xeon E5-2640 CPU running at 2.40 GHz with 64 GB of RAM and a 10G ConnectX4-L NIC. These servers are connected via a Mellanox SN240 10 Gbps TOR switch. We used a second cluster with 4 servers to perform experiments at 100 Gbps. Each server has an AMD EPYC 7452 64-Core CPU running at 2.30 GHz with 128 GB of RAM and a 100 Gbps ConnectX-5 NIC. These servers are connected via a Dell Z9264F-ON switch.

Applications: When experimenting with TCP, we leveraged the TAS TCP acceleration service to connect three unmodified POSIX applications to Backdraft: Memcached [7], Mutilate [64], and a custom distributed application that performs RPCs. To perform experiments with Homa, we utilized the Homa DPDK implementation [4], which unfortunately does not have any native support for applications. We overcome this problem by developing an open-loop RPC application on top of Homa. Because PicNIC [61] is proprietary software, a head-to-head comparison is not feasible.

Performance metrics and comparison points: Our experiments focus on four main metrics: packet drop rate, CPU utilization, throughput, and 99th percentile request completion time latency. We also compared Backdraft against two variations of BESS virtual switch: lossy (default), and a lossless variation which generates PFC messages.

Key results: With Backdraft, the Homa-based RPC application achieves 20x lower tail latency at the 99th percentile (§6.1). Further, Memcached achieves 1.9x higher goodput with Backdraft (§6.2). In a lossless multi-node scenario, Backdraft prevents congestion spreading in the network core (§6.3). In a 100 Gbps setup, Backdraft avoids HOL blocking and reaches 100 Gbps even in presence of slow receivers (§6.4). Finally, Backdraft supports 16 K queues without any throughput slow down (§6.5).

6.1 Backdraft Complements Homa

Our first experiment demonstrates that Backdraft complements Homa. In this experiment, we used two different machines in the 100 Gbps cluster: one of them hosting three client applications and the other two server applications. Each client/server application is assigned to a single CPU core. We used two clients to generate fixed-size RPC requests towards one server. The other client, instead, generates requests to

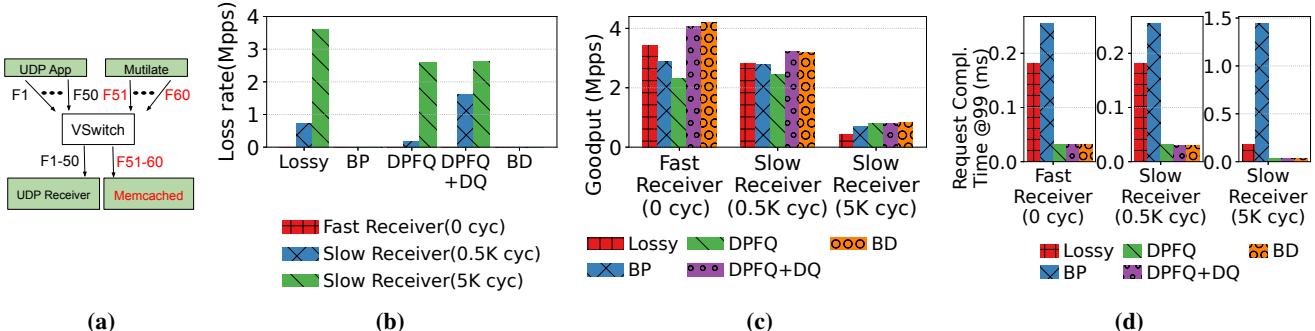


Figure 10: Performance of the individual components of Backdraft in presence of slow receivers when handling a Memcached TCP incast (10 flows) workload with a background UDP workload (50 flows). (a) Experimental setup (b) Aggregate drop rate, when the UDP server spends on average 0/0.5 K/5 K extra cycles on every delivered packet. Slower receivers have more detrimental impact on the performance. (c,d) detailed breakdown of goodput and latency impact of Backdraft. Backdraft improves tail latency up to 5.65x compared to BESS, and 45.2x compared to BESS augmented with PFC at the 99th percentile while achieving 1.9x higher goodput.

wards the remaining server using the Facebook Memcached workload [15].

We compare the RPC performance of the client using the Memcached workload when using either BESS or Backdraft as vswitch. In Figure 9a and Figure 9b, we show the results when fixed-size RPCs are 200 B and 5 KB, respectively. When the RPC load increases, the completion time with Backdraft remains stable, while it inflates by over 20x with BESS. The poor results experienced with BESS are a consequence of its single queue design. In contrast, Backdraft keeps tail latency low because each `RPC_ID` occupies a single queue in the vswitch. This way, Backdraft removes HOL blocking of various RPCs with different service times.

Homa and Backdraft have strong synergy. Homa eagerly sends RESEND control messages to peers (`RESEND_INTERVAL = 2 μs` [5]). This enables Homa to detect packet loss proactively, resulting in better tail latency. The CPU overhead of this task can be prohibitively high in presence of packet loss. For instance, without Backdraft, the CPU usage of Homa increases 8-10% because there are more outstanding messages to manage due to loss. Backdraft avoids wasting CPU cycles by preventing packet loss, enabling the transport protocol to provide better performance.

6.2 Per-Component Analysis

To provide a performance breakdown of the benefits of the different Backdraft components, we created a scenario in a single host where background UDP packets destined to a slow receiver (50 flows) compete against a Memcached application with 10 active flows generated by Mutilate (Figure 10a). Here, we considered three cases: (1) the receiver spends 0 cycles processing the received packet; (2) the receiver spends 500 cycles; and (3) the receiver spends 5000 cycles. For context, Facebook’s Katran load balancer spends 100 cycles per packet [20], and complex functions like range queries in key-value stores can easily take more than 1 K cycles.

Figure 10 shows the results of this experiment. With *Lossy*, we consider the default behavior of BESS, while

BP is BESS with PFC enabled. DPFQ, DPFQ+DQ, and DPFQ+DQ+ON (BD) show the incremental benefits of different Backdraft components: dynamic per-flow input queueing (DPFQ), doorbell queues (DQ), and the overlay network (ON). BD indicates our final system with all components.

Figure 10b shows packet loss rates given the slow receiver application (UDP receiver) in Figure 10a. BESS with PFC and Backdraft both report zero packet loss. Without PFC for BESS and without the overlay network for Backdraft, packets may be dropped. Packet loss occurs in both the slow and fast flows, and it is more problematic in the presence of a slow receiver. DPFQ+DQ reduces CPU overheads and can forward at higher throughputs than just DPFQ. This results in even more packet loss at the receiver. This packet loss, however, is avoided by introducing the overlay network (ON). Backdraft prevents packet loss and achieves higher throughput and lower tail latency.

Next, Figure 10c shows the aggregate goodput achieved by the applications (UDP and Memcached). Backdraft always outperforms BESS, even when the latter is augmented with lossless capabilities using PFC. In this experiment, for the 0, 500, and 5 K cycle receiver, Backdraft achieves 22%, 10%, and 200% and higher goodput than the lossy counterpart, respectively. Backdraft also mitigates tail latency at the 99th percentile by up to 5.65x.

Looking at the individual components, we find that DPFQ has a negative impact on performance because polling more queues consumes more CPU cycles. However, combining DPFQ and doorbell queues (DPFQ+DQ) improves goodput by reducing cycles spent polling. This effect is more visible in the presence of a fast receiver, as the faster the receiver the more packets need to be processed by the vswitch. The last component (ON) enables Backdraft to prevent packet loss. This is illustrated in Figure 10b.

Figure 10d shows the latency experienced by Memcached. In this figure, Backdraft similarly outperforms both BESS configurations. The *BP* bar shows that naively applying a backpressure mechanism dramatically increases network latency, and this is mainly an effect of HOL blocking. DPFQ, in

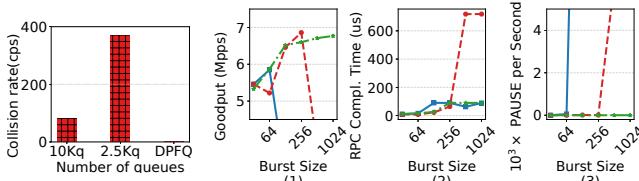


Figure 11:

Compares collision rate of a skewed workload when varying the number of queues.

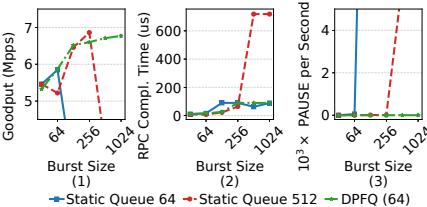


Figure 12: Shows goodput, latency and PAUSE rate of short queues, long queues verses DPFQ. Backdraft can absorb different burst sizes compared to static queue allocation with PFC.

contrast, keeps the overall latency low, even in the presence of a slow receiver. This is because providing per-flow queues prevents HOL blocking.

To better understand Dynamic Queue Allocation (DQA), we used a sample client application generating approximately 100 K flows to a server sink application on the same machine where only 1 K flows are active at any point in time. We compared two different policies for queue allocation: a static number defined at configuration time and a dynamic. The former assigns flows to queues using an RSS (Receive Side Scaling) hash function, the latter creates a new queue anytime a new flow shows up. Figure 11 shows that when using only 2.5 K queues, the collision rate is high, even if only 1 K flows are active. Having 10x more static queues than active flows helps, but still collisions occur. In contrast, DPFQ avoids wasted memory and achieves a zero-collision rate thanks to its per-flow queueing mechanism. Each ring buffer consumes about 20 B. 10 K queues will consume 200 KB, where DPFQ allocates only 1 K queues since we have 1 K active flows, requiring only 20 KB. This is a 10x reduction in memory utilization in addition to the reduction in collisions.

Next, we evaluated Backdraft’s ability to absorb packet bursts by extending queues by performing an experiment where a sender pushes different batch sizes (64 to 1024) to a receiver. The receiver is attached to a vswitch on the same server and pulls packets in large batches of 1024. This experiment compares Backdraft against two different configurations of BESS augmented with PFC: one with short queues, the other with long. Short queues are more likely to generate PAUSE frames at a higher rate, whereas longer queues are less likely.

Figure 12 shows that, when increasing the burst size, lossless BESS with short queues causes a high PFC PAUSE frame generation rate that would hurt application performance in terms of goodput and tail latency. Although long queues reduce the PAUSE frame generation problem, this is at the cost of increased latency. In contrast, this experiment shows that Backdraft is able to absorb variations, particularly in a bursty workload with its dynamic queue extensions. It is the only configuration that does not generate PAUSE frames. Moreover, Backdraft maintains high goodput with DPFQ because the cost of queue extension is relatively low.

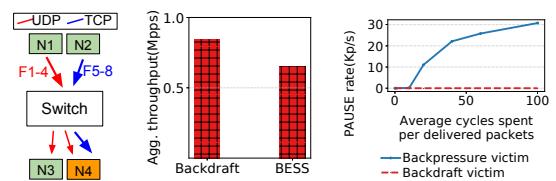


Figure 13: Performance of Backdraft overlay network in a cluster-wide experiment. Backdraft achieves higher throughput and avoids extra PAUSE frame generation in presence of a slow receiver. (a) Experiment setup. (b) Overall throughput. (c) Pause frame generation rate due to a slow receiver.

Config	Tput (Gbps)	Pause (Kfps)	Drop (Mpps)
BESS Lossy + Lossy Network	(2.36, 21.85)	N/A	(1.6, 0)
BESS Lossless + Lossy Network	(2.66, 19.29)	(2.8, 0)	(1.3, 0)
ON + Lossy Network	(2.3, 21.98)	(0, 0)	(0, 0)

Table 3: Virtual overlay network performance results (Victim, Non-victim flow)

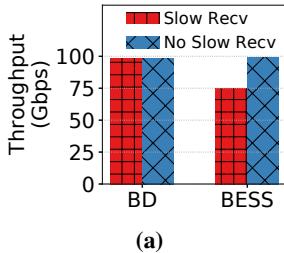
Finally, we also measured the overheads of extending queues in DPFQ and found that it is small. The number of cycles required to extend queues fluctuates between 24 and 350 cycles, and this value is dependent on caching. This shows that the overheads of DPFQ are low, especially when amortized over all of the packets in the added queue, which has a default size of 64 packets. Further, if desired, Backdraft’s queue size can be configured as a parameter based on the measured overhead according to the user’s preference for performance versus memory efficiency.

6.3 Multi-node Performance

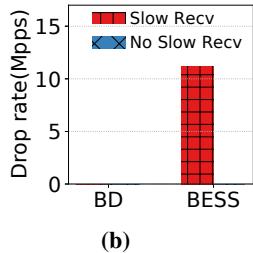
This section studies the behavior of the overlay network between vswitches used in Backdraft. To do this, we used a cluster of four different servers. Each server is running its own vswitch, and they are connected through a physical switch with PFC enabled. We generated background UDP flows competing with TCP victim flows (Figure 13a) and compared the results when using either BESS or Backdraft as a vswitch. Backdraft achieves higher aggregate throughput than BESS (Figure 13b). This is because Backdraft sends PAUSE frames through the overlay networks as soon as it notices queue buildup. This is not done by BESS, which in turn induces the physical switch to send PFC PAUSE frames and trigger congestion spreading inside the network.

Figure 13c shows the number of PFC PAUSE frames sent by the receiving server to the upstream PFC enabled switch as a receiver gets slower. In this scenario, BESS causes the physical switch to also generate PAUSE frames. However, this does not happen with Backdraft because the overlay network pauses the flow for the slow receiver before queues fill up.

We also compared the performance of BESS and Backdraft using two nodes in the 100 Gbps CloudLab testbed connected



(a)



(b)

Figure 14: Multithreading in Backdraft, (a) Aggregate throughput, (b) Drop rate of victim flow. Backdraft achieves 100 Gbps using multiple cores while ensuring zero drop at the vswitch. In this experiment, applications are allocated enough number of cores to drive 100 Gbps.

by a lossy switch. Here, we used one server to send two UDP flows towards another machine where one receiver is slow and the other instead is a victim. Table 3 reports our results. When using standard BESS (lossy) with a lossless network core (first row), the overall throughput is high. However, it also suffers from a high degree of packet loss. When, instead, using BESS generating PFC frames (lossless), the throughput is reduced and a considerable amount of packet loss still appear, as the network core is lossy. Finally, due to overlay messages, Backdraft it is able to avoid packet losses, while keeping network throughput high.

Finally, we performed an experiment to demonstrate that the overlay network in Backdraft does not suffer from starvation, even when the rate of the slow receiver is variable over the time. In this experiment, one machine is sending packets towards a slow receiver. Initially, the destination polls packets at rate 3 Mpps, then it doubles its rate at time T_{30} . In Figure 15, BP (BESS with PFC) suffers from starvation and the receiver spends its extra cycles polling instead of processing packets. In contrast, at $T = T_{30}$, Backdraft detects a change in the receivers rate and increases Th_{goal} to avoid starvation for the rest of the application’s life.

6.4 100 Gbps Forwarding Performance

To show that Backdraft can achieve 100 Gbps throughput regardless of the presence of slow receivers, we performed an experiment where an 8-core sender is generating a heavily skewed workload consisting of 12 flows (11 fast flows and 1 slow flow) towards an 8-core receiver. To cause a slow flow, one of 8 cores of the receiver application is slowed down in this experiment. When using BESS, the slow flow will eventually block the others, forcing the vswitch to drop packets due to a lack of queue descriptors at the receiver’s RX queues. In contrast, Backdraft does not suffer from this problem because of its ability to dynamically resize queues and send overlay PAUSE frames.

Figure 14 shows the aggregate throughput for all flows and the drop rate for the victim flow in presence of slow receivers in this experiment. With BESS, this experiment results in high packet loss and a decreased throughput of ~ 75 Gbps

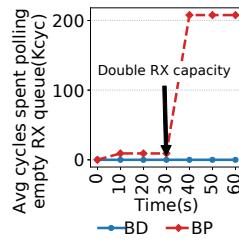


Figure 15: Backdraft has no starvation. Backdraft adjusts the sender rate using overlay messages to ensure enough buffering at the receiver’s queues.

even though there is only one core receiving slower than the expected pace. In contrast, Backdraft achieves full line-rate without any packet drops. Backdraft sends overlay messages on a per-queue basis to notify the upstream sender to reduce its rates. This allows Backdraft to utilize the extra bandwidth for the other flows in order to drive the 100 Gbps line-rate.

6.5 Backdraft Scalability

Finally, we assessed the scalability of Backdraft in terms of its throughput and memory requirements.

The impact of number of queues on performance. To demonstrate the benefits of doorbell queues, we performed an experiment where an application sends packets from UDP flows in a skewed pattern based on the Zipfian distribution, and we compared the throughput achieved between doorbell queues (Backdraft) and polling every queue (BESS).

Figure 16 shows the aggregate vswitch throughput when a single core is allocated to the switch as we vary the number of queues. With a small number of queues, both Backdraft and BESS perform similarly, which shows that the overheads of doorbell queues are quite low. However, when the number of queues increase, only Backdraft maintains its throughput.

The amount of memory needed varying network RTT. Finally, we performed an analysis of the memory overheads of Backdraft to demonstrate that this is not prohibitive. In order to avoid congestion spreading, Th_{net} must be sufficiently larger when compared to Th_{over} so that packets can be buffered during the time it takes for the source of the congestion to pause and adjust its rate. The increased memory overheads of Backdraft are small and can be estimated by bandwidth-delay product for different network line-rates. For example, a 100 Gbps network with a 1ms RTT only requires 12.5 MB of buffering to avoid congestion spreading. Further, it is important to note that DPFQ ensures that this buffering requirement is for the entire switch and not per-flow.

7 Discussion

Slow NICs. NICs may be slow and unable to achieve line-rate, and this can cause packet loss [44, 83]. If a slow NIC

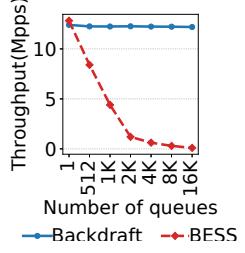


Figure 16: Backdraft, compared to BESS, sustains high throughput with a single core while managing large number of queues as it mitigates the polling overhead.

participates in the overlay network by generating OPFs, it can avoid both packet loss and congestion spreading.

Slow virtual switches. There are many reasons a vswitch may be slow, including CPU limitations, memory bandwidth limitations, and insufficient LLC cache [34, 36, 89, 95], and packets can be dropped at the NIC when a vswitch cannot keep up with the ingress rate, resulting in a slow vswitch problem. This can be solved by offloading part of Backdraft’s processing onto a programmable NIC [2, 9, 39, 46, 87]. This should be feasible because recent developments in NIC designs have brought models that provide a large amount of on-NIC memory that can be used for Backdraft. For example, Xilinx Alveo NICs support High Bandwidth Memory (HBM), fast memory that is directly embedded on-chip in an FPGA [11, 53].

RDMA support. Backdraft can support 2-sided RDMA verbs by monitoring the length of receive queues in the application or a library and generating OPF messages to transmitters as appropriate. Further, if offloaded onto a NIC, Backdraft can mitigate the effect of the slow NIC problem for 1-sided verbs and complement the sender-based approaches that can be used for congestion control [55, 68, 98].

Programmable packet scheduling. If Backdraft is deployed without enough memory, multiple flows have to share the same queue. Although this can cause HOL blocking, this can be mitigated with opportunistic packet scheduling. For example, Backdraft could employ software solutions like Eiffel [80] or hardware ones like AIFO [94] and PIFO [85] if Backdraft is offloaded to a programmable NIC.

Linux kernel compatibility. Backdraft is implemented using DPDK. Thus, all of the traffic coming from the NIC bypasses the Linux kernel. However, we believe that the same design principles are applicable to the Linux kernel. Further, being implemented in userspace does not even preclude Backdraft from interfacing with the Linux kernel networking stack. For example, Backdraft can use a custom kernel driver to interface with traditional applications, and the recently proposed AF_XDP Poll Mode driver [1] enables DPDK applications to natively support the AF_XDP socket and retain compatibility with the Linux tools that operators expect [90].

8 Related Work

Slow receiver problem. Past research has acknowledged the slow receiver problem in the context of the overheads of the Linux networking stack [21], Linux-based transport protocol implementations [73], and production networks from Microsoft [44], and Google Swift [60].

Virtual switching. Snap, Andromeda, and PicNIC all perform lossy vswitching [30, 61, 68], which drops packets. On the other hand, Zfabric, NFNice, and zOVN are lossless vswitches. These, however, suffer from HOL blocking as they share queues among active flows in the vswitch [26, 27, 59]. Moreover, unlike Backdraft, none of these approaches address

the slow receiver problem. Similarly, FreeFlow ensures high performance by using shared memory, but it does not consider packet loss problem at the end-hosts [93].

Packet scheduling and rate limiting. Backdraft is compatible with Eiffel and Carousel and can mitigate their CPU utilization overheads with its command queue [79, 80]. Similarly, hyperplane can be used to reduce the CPU polling overheads of Backdraft [70]. EyeQ is a related system that builds an overlay network that performs rate-limiting [50]. However, EyeQ pays high CPU utilization overhead when rate limiting, and EyeQ works at millisecond-scale, which is not fast enough to address the slow receiver problem.

Congestion control. In addition to Homa, there are other important new congestion control algorithms like Google’s Swift, which performs fine grain time stamping to identify the congestion source (end-host, network) [60]. Similar to how we have found that Backdraft is complementary to Homa, we expect that Backdraft is complementary to Swift as well.

Flow control. Backdraft is complementary to flow control protocols designed to provide a lossless core network. For example, Backdraft is complementary to PFC because it strives to minimize the PAUSE frames sent across the network. PCN ensures high throughput for victim flows if congestion spreading occurs and is also complementary to Backdraft [23]. BFC is a new backpressure flow control protocol intended to replace PFC [43]. Backdraft solves a key problem that arises with deploying BFC in practice. This is because BFC assumes that flows can be received at 100 Gbps line-rates, and this assumption can be violated by slow receivers. Backdraft addresses this problem and prevents congestion spreading from slow receivers.

9 Conclusions

In this paper, we present the design and implementation of Backdraft, a new lossless virtual switch. We make a case for providing lossless networking at the vswitch level by showing the impact of packet loss caused by slow receivers on network performance using existing congestion control algorithms.

We implemented Backdraft on top of the BESS virtual switch and performed experiments with two different clusters of servers on CloudLab (10 Gbps and 100 Gbps). We used unmodified POSIX applications with TAS TCP and a custom distributed application that performs RPCs with Homa, a state-of-the-art datacenter transport protocol. We demonstrate that Backdraft is effective in preventing packet loss and reduces tail latency by up to 20x compared to BESS.

Acknowledgements: We thank our shepherd, Anurag KhanDELWAL, the anonymous NSDI reviewers, Praveen Kumar, and Djordje Jevdjic for their feedback. This work was funded by NSF Awards CNS-2200783 and CNS-2008273, the UK EPSRC project EP/T007206/1, and by gifts from Google and VMware.

References

- [1] Af-xdp poll mode driver. https://doc.dpdk.org/guides/nics/af_xdp.html.
- [2] Alveo SN1000 SmartNIC Accelerator Card. <https://www.xilinx.com/products/boards-and-kits/alveo/sn1000.html>.
- [3] BESS: Berkeley Extensible Software Switch. <https://github.com/NetSys/bess>.
- [4] Homa. <https://github.com/PlatformLab/Homa>.
- [5] Homa commit 47265bf. <https://github.com/PlatformLab/Homa/blob/392b57bbdad2f5aa42faefc88614992b5e505d2/src/TransportImpl.cc#L36>.
- [6] iperf3: Documentation. <http://software.es.net/iperf/>.
- [7] Memcached. <https://memcached.org/>.
- [8] Nginx. <https://www.nginx.com/>.
- [9] Virtual Switch on BlueField SmartNIC. <https://docs.mellanox.com/display/BlueFieldSWv20110841/Virtual+Switch+on+BlueField+SmartNIC>.
- [10] What's New in Virtio 1.1. https://www.dpdk.org/wp-content/uploads/sites/35/2018/09/virtio-1.1_v4.pdf.
- [11] Xilinx alveo u280. <https://www.xilinx.com/products/boards-and-kits/alveo/u280.html>.
- [12] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2010.
- [13] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2012.
- [14] N. Amit, A. Tai, and M. Wei. Don't shoot down tlb shootdowns! In *European Conference on Computer Systems (EuroSys)*. ACM, 2020.
- [15] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. ACM, 2012.
- [16] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. In *Communications of the ACM*, volume 60, pages 48–54. ACM, 2017.
- [17] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013.
- [18] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2010.
- [19] J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, and J. Sopena. The battle of the schedulers: Freebsd ULE vs. linux CFS. In *Annual Technical Conference (ATC)*. USENIX, 2018.
- [20] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco. hXDP: Efficient software packet processing on FPGA nics. In *Operating Systems Design and Implementation (OSDI)*. USENIX, 2020.
- [21] Q. Cai, S. Chaudhary, M. Midhul, Vuppalaipati, J. Hwang, and R. Agarwal. Understanding Host Network Stack Overheads. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2021.
- [22] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *Workshop on Research on Enterprise Networking (WREN)*. ACM, 2009.
- [23] W. Cheng, K. Qian, W. Jiang, T. Zhang, and F. Ren. Re-architecting congestion management in lossless Ethernet. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2020.
- [24] I. Cho, K. Jang, and D. Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.
- [25] C. Chou, L. N. Bhuyan, and D. Wong. μ dpm: Dynamic power management for the microsecond era. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019.
- [26] D. Crisan, R. Birke, N. Chrysos, C. Minkenberg, and M. Gusat. zFabric: How to virtualize lossless Ethernet? In *International Conference On Cluster Computing (CLUSTER)*. IEEE, 2014.
- [27] D. Crisan, R. Birke, G. Cressier, C. Minkenberg, and M. Gusat. Got loss? get zOVN! In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2013.

- [28] A. Daglis, M. Sutherland, and B. Falsafi. RPCValet: Ni-driven tail-aware balancing of μ s-scale RPCs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2019.
- [29] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. In *Transactions on Computers*, volume 36, pages 547–553. IEEE, 1987.
- [30] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2018.
- [31] C. DeSanti. IEEE 802.1: 802.1Qbb - Priority-based Flow Control. <http://www.ieee802.org/1/pages/802.1bb.html>, 2009.
- [32] D. Didona and W. Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.
- [33] C. Fang, H. Liu, M. Miao, J. Ye, L. Wang, W. Zhang, D. Kang, B. Lyv, P. Cheng, and J. Chen. Vtrace: Automatic diagnostic system for persistent packet loss in cloud-scale overlay network. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2020.
- [34] A. Farshin, T. Barbette, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić. PacketMill: Toward per-core 100-gbps networking. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2021.
- [35] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostić. Reexamining direct cache access to optimize i/o intensive applications for multi-hundred-gigabit networks. In *Annual Technical Conference (ATC)*. USENIX, 2020.
- [36] A. Farshin, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In *Annual Technical Conference (ATC)*. USENIX, 2020.
- [37] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaei, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Special Interest Group on Programming Languages (SIGPLAN)*. ACM, 2012.
- [38] D. Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2017.
- [39] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2018.
- [40] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2020.
- [41] H. Golestani, A. Mirhosseini, and T. F. Wenisch. Software data planes: You can't always spin to win. In *Symposium on Cloud Computing (SoCC)*. ACM, 2019.
- [42] R. Gouicem, D. Carver, J.-P. Lozi, J. Sopena, B. Lepers, W. Zwaenepoel, N. Palix, J. Lawall, and G. Muller. Fewer cores, more hertz: Leveraging high-frequency cores in the OS scheduler for improved application performance. In *Annual Technical Conference (ATC)*. USENIX, 2020.
- [43] P. Goyal, P. Shah, K. Zhao, G. Nikolaidis, M. Alizadeh, and T. E. Anderson. Backpressure flow control. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2022.
- [44] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.
- [45] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A software nic to augment hardware. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155*, 2015.
- [46] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *Workshop on Hot Topics in Networks (HotNets)*. ACM, 2019.
- [47] D. Intel. Data plane development kit, 2014.
- [48] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. Perfiso: Performance isolation for commercial latency-sensitive services. In *Annual Technical Conference (ATC)*. USENIX, 2018.

- [49] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable Message Latency in the Cloud. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2015.
- [50] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2013.
- [51] K. Kaffles, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μ second-scale tail latency. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.
- [52] K. Kaffles, J. T. Humphries, D. Mazières, and C. Kozyrakis. Syrup: User-defined scheduling across the stack. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 2021.
- [53] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso. High bandwidth memory on FPGAs: A data analytics perspective. In *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020.
- [54] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2015.
- [55] G. P. Katsikas, T. Barbette, M. Chiesa, D. Kostić, and G. Q. Maguire. What you need to know about (smart) network interface cards. In *International Conference on Passive and Active Network Measurement*. Springer, 2021.
- [56] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP acceleration as an OS service. In *European Conference on Computer Systems (EuroSys)*. ACM, 2019.
- [57] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Annual Technical Conference (ATC)*. USENIX, 2019.
- [58] A. Kopytov. Sysbench: a system performance benchmark. <http://sysbench.sourceforge.net/>, 2004.
- [59] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.
- [60] G. Kumar, N. Dukkipati, K. Jang, H. M. G. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall, and A. Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2020.
- [61] P. Kumar, N. Dukkipati, N. Lewis, Y. Cui, Y. Wang, C. Li, V. Valancius, J. Adriaens, S. Gribble, N. Foster, and A. Vahdat. Picnic: Predictable virtualized nic. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2019.
- [62] H. T. Kung, T. Blackwell, and A. Chapman. Credit-based flow control for ATM networks: Credit update protocol, adaptive credit allocation and statistical multiplexing. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 1994.
- [63] B. Lepers, R. Gouicem, D. Carver, J.-P. Lozi, N. Palix, M.-V. Aponte, W. Zwaenepoel, J. Sopena, J. Lawall, and G. Muller. Provable multicore schedulers with ipanema: Application to work conservation. In *European Conference on Computer Systems (EuroSys)*. ACM, 2020.
- [64] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *European Conference on Computer Systems (EuroSys)*. ACM, 2014.
- [65] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Symposium on Cloud Computing (SoCC)*. ACM, 2014.
- [66] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2015.
- [67] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry. Contention-aware performance prediction for virtualized network functions. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2020.
- [68] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, et al. Snap: a microkernel approach to host networking. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 2019.
- [69] A. Menon and W. Zwaenepoel. Optimizing TCP receive performance. In *Annual Technical Conference (ATC)*. USENIX, 2008.
- [70] A. Mirhosseini, H. Golestani, and T. F. Wenisch. Hyperplane: A scalable low-latency notification accelerator for software data planes. In *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2020.

- [71] A. Mirhosseini, B. L. West, G. W. Blake, and T. F. Wenisch. Q-zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020.
- [72] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2018.
- [73] J. Ousterhout. A linux kernel implementation of the homa transport protocol. In *Annual Technical Conference (ATC)*. USENIX, 2021.
- [74] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: Core-aware thread management. In *Operating Systems Design and Implementation (OSDI)*. USENIX, 2018.
- [75] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 2014.
- [76] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *Security Symposium (USENIX Security)*. USENIX, 2012.
- [77] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network’s (Datacenter) Network. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2015.
- [78] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. In *SIGOPS Operating Systems Review*, volume 42, pages 95–103. ACM, 2008.
- [79] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.
- [80] A. Saeed, Y. Zhao, N. Dukkipati, E. Zegura, M. Ammar, K. Harras, and A. Vahdat. Eiffel: Efficient and flexible software packet scheduling. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.
- [81] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *Operating Systems Design and Implementation (OSDI)*. USENIX, 2018.
- [82] E. Sharafzadeh, A. Sanaee, E. Asyabi, and M. Sharifi. Yawn: A cpu idle-state governor for datacenter applications. In *SIGOPS Asia-Pacific Workshop on Systems (APSys)*. ACM, 2019.
- [83] A. Singhvi, A. Akella, D. Gibson, T. F. Wenisch, M. Wong-Chan, S. Clark, M. M. K. Martin, M. McLaren, P. Chandra, R. Cauble, H. M. G. Wassel, B. Montazeri, S. L. Sabato, J. Scherpelz, and A. Vahdat. 1RMA: Re-envisioning remote memory access for multi-tenant datacenters. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2020.
- [84] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.
- [85] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.
- [86] A. Sriraman and A. Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2020.
- [87] B. Stephens, A. Akella, and M. Swift. Loom: Flexible and efficient NIC packet scheduling. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.
- [88] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. Practical DCB for improved data center networks. In *Conference on Computer Communications (INFOCOM)*. IEEE, 2014.
- [89] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker. ResQ: Enabling SLOs in network function virtualization. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2018.
- [90] W. Tu, Y.-H. Wei, G. Antichi, and B. Pfaff. Revisiting the open vswitch dataplane ten years later. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2021.
- [91] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2009.
- [92] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Operating Systems Design and Implementation (OSDI)*. USENIX, 2020.

- [93] T. Yu, S. A. Noghabi, S. Raindel, H. Liu, J. Padhye, and V. Sekar. FreeFlow: High Performance Container Networking. In *Workshop on Hot Topics in Networks (HotNets)*. ACM, 2016.
- [94] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin. Programmable packet scheduling with a single queue. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2021.
- [95] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim. Don't forget the I/O when allocating your LLC. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2021.
- [96] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the flow completion time tail in datacenter networks. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2012.
- [97] X. Zhan, R. Azimi, S. Kanev, D. Brooks, and S. Reda. Carb: A c-state power management arbiter for latency-critical workloads. In *Computer Architecture Letters*, volume 16, pages 6–9. IEEE, 2016.
- [98] Y. Zhang, Y. Tan, B. Stephens, and M. Chowdhury. Justitia: Software multi-tenancy in hardware kernel-bypass networks. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2022.
- [99] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale RDMA deployments. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2015.

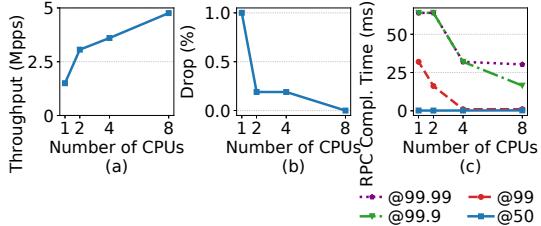


Figure 17: Throughput, loss and latency of DCTCP given different number of allocated cores. DCTCP still is unable to prevent packet loss with 4 CPU cores.

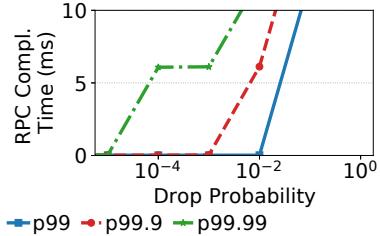


Figure 18: Homa experiences millisecond scale tail latency with even 10^{-2} drop probability.

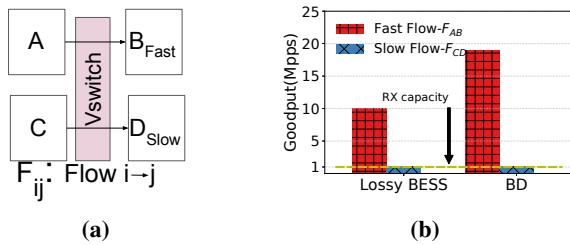


Figure 19: Backdraft TX bandwidth management in presence of a slow receiver. (a) The experiment setup on a single machine. (b) Backdraft prevents packet loss and saves CPU utilization from slow receiver and can allocate it to other receiver applications.

A Appendices

In this section, we expand our experiments associated with congestion controls, and bandwidth management on a single host.

A.1 Slow Receivers and DCTCP/Homa

While we discussed the problem associated with congestion controls such as Homa with regard to the slow receiver problem in Figure 4, we extended our study and performed similar experiments on DCTCP (§A.1.1).

We then discuss the impact of packet loss on latency of Homa (§A.1.2), given that Homa uses high granular timers to identify lost packets which is already discussed in §6.1.

A.1.1 DCTCP

We show that congestion control algorithms fail to address slow receiver problem in §2.1. Other than Homa, we per-

formed the same test on DCTCP congestion control. Figure 17a show that throughput of DCTCP application cannot reach higher than 5 Mpps or 320 Mbps with even 8 cores (64 B packets were used).

We have found that this packet loss occurs even when the vswitch performs ECN marking and end-hosts use a state-of-the-art congestion control algorithm like DCTCP [12]. This is demonstrated in Figure 17b, which shows what happens when we vary the number of allocated cores from 1 to 8 allocated to a DCTCP receiver application experiencing receiving data from a DCTCP client that is utilizing 8 CPU cores to send messages as fast as possible. We enable ECN marking at *vswitch level* to ensure DCTCP controls the flow rates in the scenarios where only vswitches are involved. Finally, Figure 17c demonstrates that packet loss has dramatic impact on the tail latency of the DCTCP.

A.1.2 Packet Loss Effect on Homa

In this section, we further discuss packet loss overhead of Homa protocol discussed in §4. In Figure 18, we observe that RPC completion time increase to 5x higher with mere packet loss probability of 10^{-2} . Although Homa identifies lost packets with high resolution timers, this does not seem to be highly effective.

A.2 Single Host Bandwidth Management

We performed an extra experiment to show how Backdraft works when dealing with a non-cooperative workload in terms of bandwidth management. This experiment is carried on a single node, we demonstrate that Backdraft delivers 2x higher throughput than its counterpart, BESS. Figure 19a shows the setup for this experiment. Here we have four applications (A, B, C , and D), where application D is a slow receiver and process packets at a maximum of 1 Mpps. The sender applications (i.e., A and C) are configured to transmit packets at 20 Mpps, instead. Reciver B is not limited in performance, so we can consider it to be fast. When Backdraft identifies the queue buildup due to slow receiver (i.e., D), it sends a local overlay message towards the sender port that includes a pause duration and an estimate of the receiver's rate. Using this information, Backdraft can pause the sender port, save CPU cycles otherwise wasted in handling the slow receiver flow, and use the saved resources to better handle the traffic directed to the fast receiver.

Figure 19b demonstrates this. With Backdraft, flow f_{AB} achieves 19 Mpps throughput. BESS, however, wastes CPU cycles and throughput bandwidth on dropping packets, causing the flow to reach only 10 Mpps.