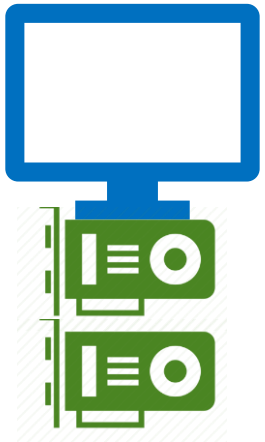
# *Quiver*: An informed storage cache for Deep Learning

**Abhishek Vijaya Kumar**, Muthian Sivathanu

*Microsoft Research India*

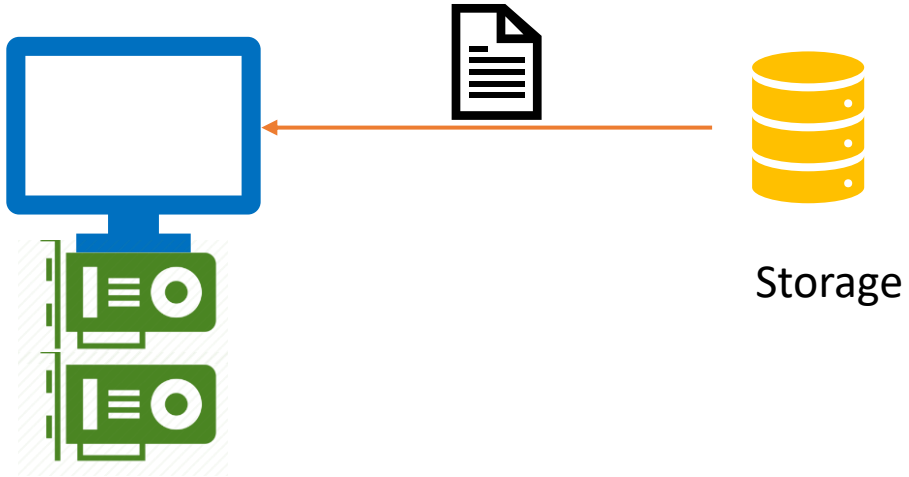
# Deep Learning: Important systems workload

- Already powers many real-world applications
  - Voice assistants
  - Web search
- Compute intensive – expensive hardware e.g. GPUs



# Deep Learning: Important systems workload

- Already powers many real-world applications
  - Voice assistants
  - Web search
- Compute intensive – expensive hardware e.g. GPUs



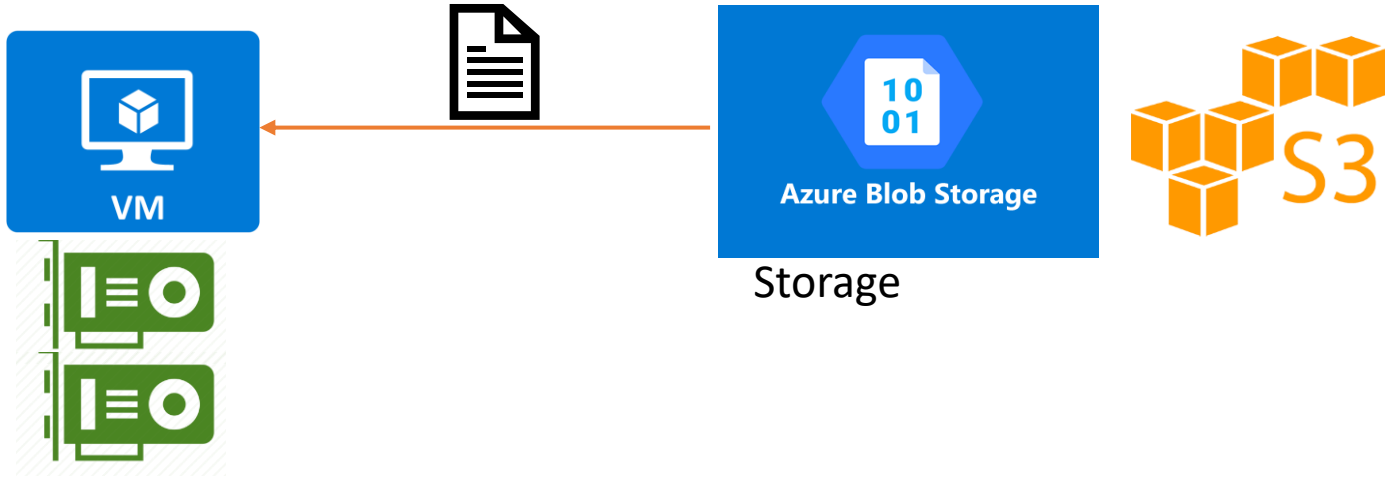
# Deep Learning: Important systems workload

- Already powers many real-world applications
  - Voice assistants
  - Web search
- Compute intensive – expensive hardware e.g. GPUs

Same setting on Cloud

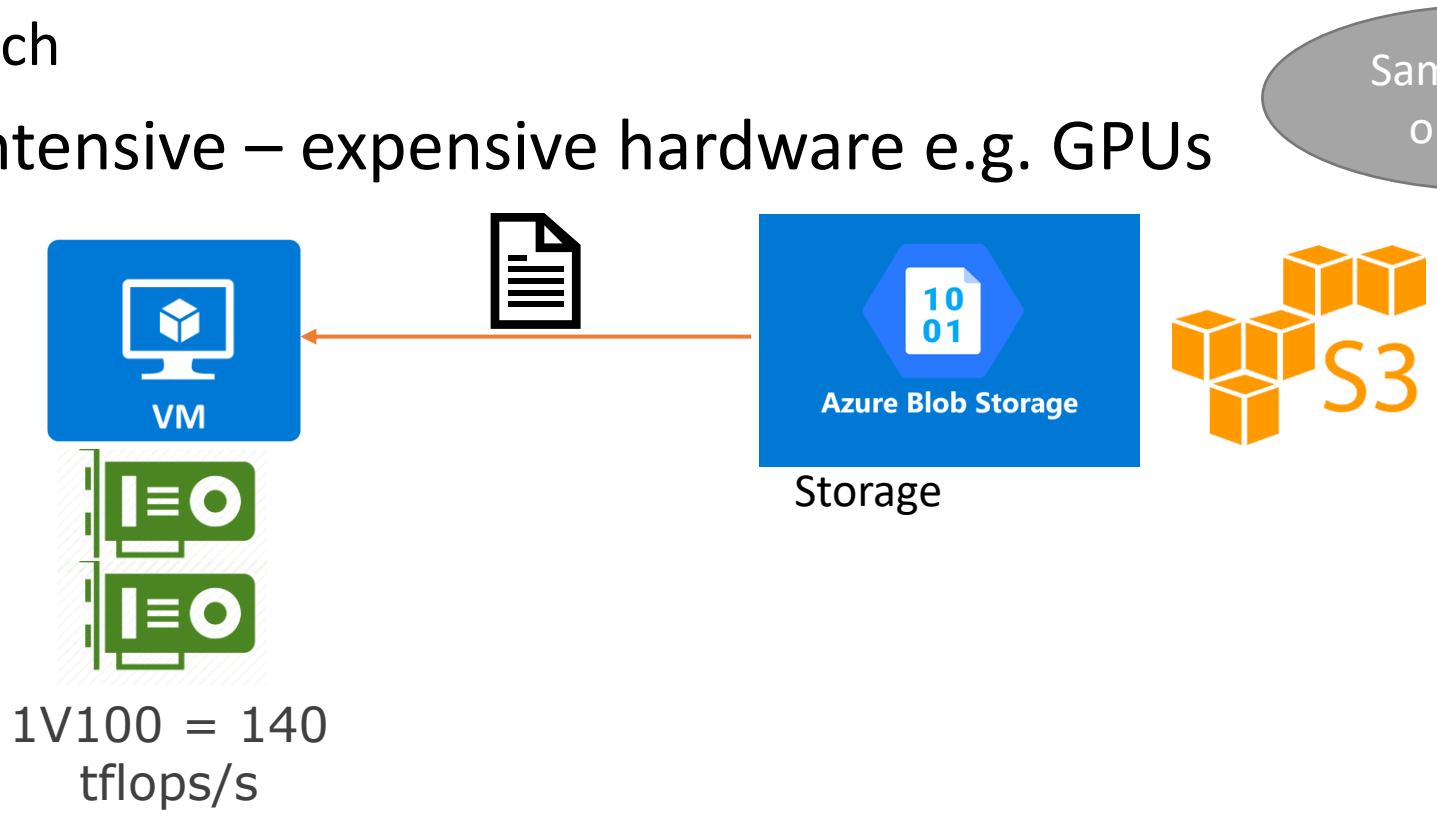
# Deep Learning: Important systems workload

- Already powers many real-world applications
  - Voice assistants
  - Web search
- Compute intensive – expensive hardware e.g. GPUs



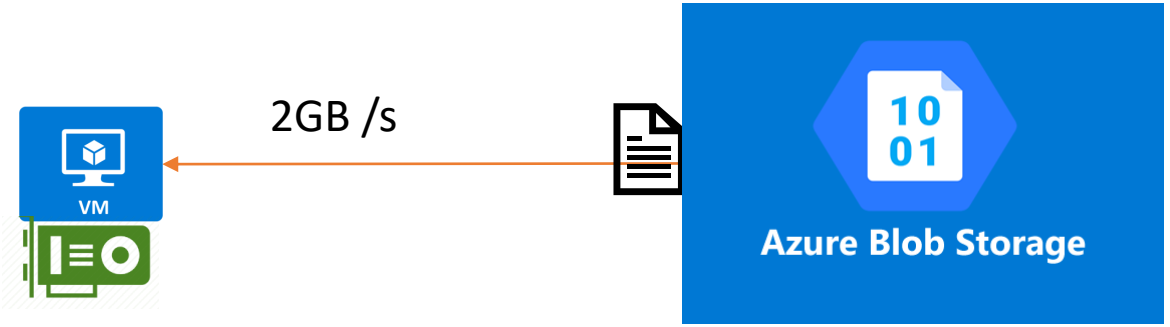
# Deep Learning: Important systems workload

- Already powers many real-world applications
  - Voice assistants
  - Web search
- Compute intensive – expensive hardware e.g. GPUs



# Example workload

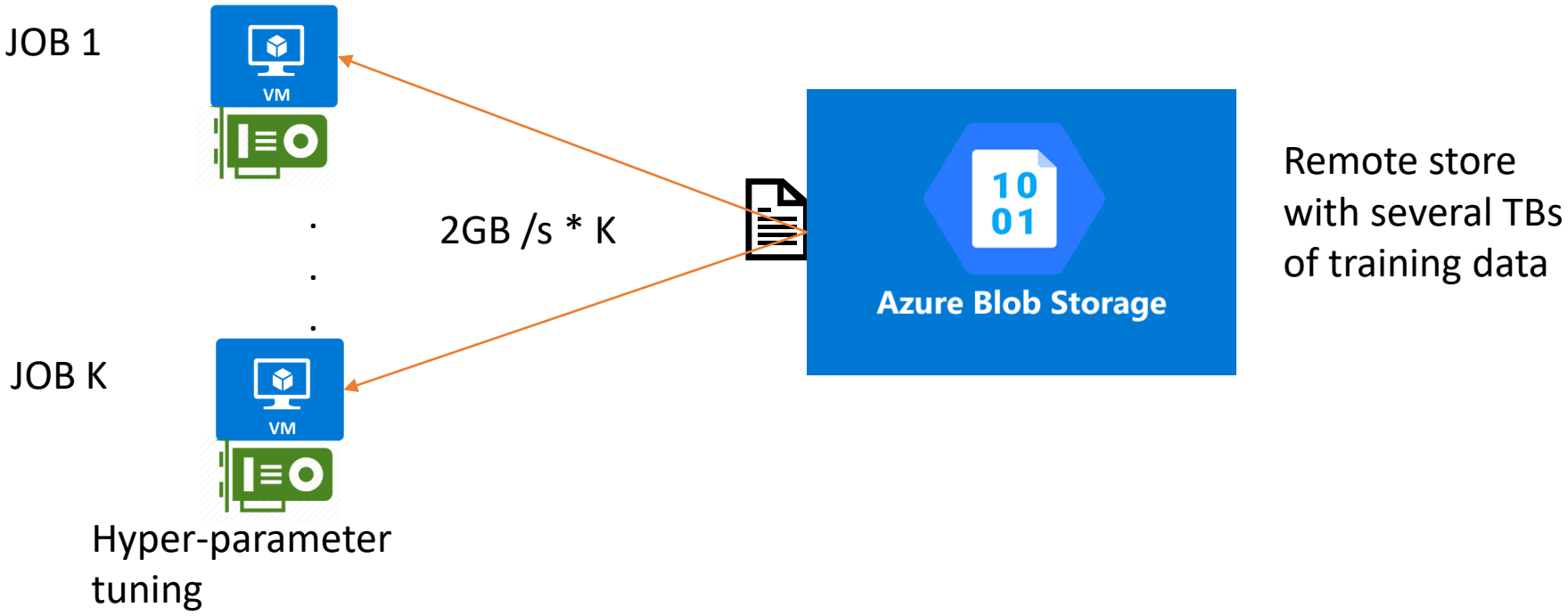
- Resnet50 is a popular vision model
- Process 10,500 images/sec on 8 Nvidia V100s
- **Goal: Keep GPUs busy and utilize them efficiently**




Remote store with several TBs of training data

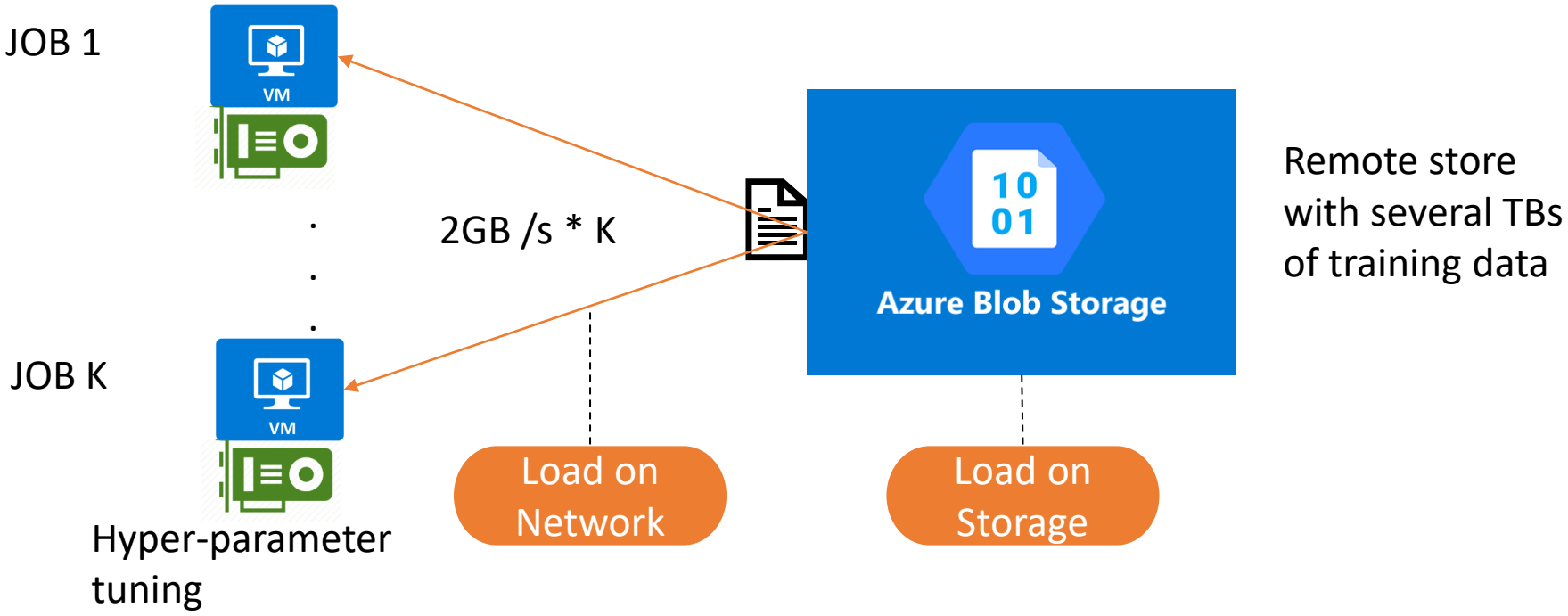
# Example workload

- Resnet50 is a popular vision model
- Process 10,500 images/sec on 8 Nvidia V100s
- **Goal: Keep GPUs busy and utilize them efficiently**



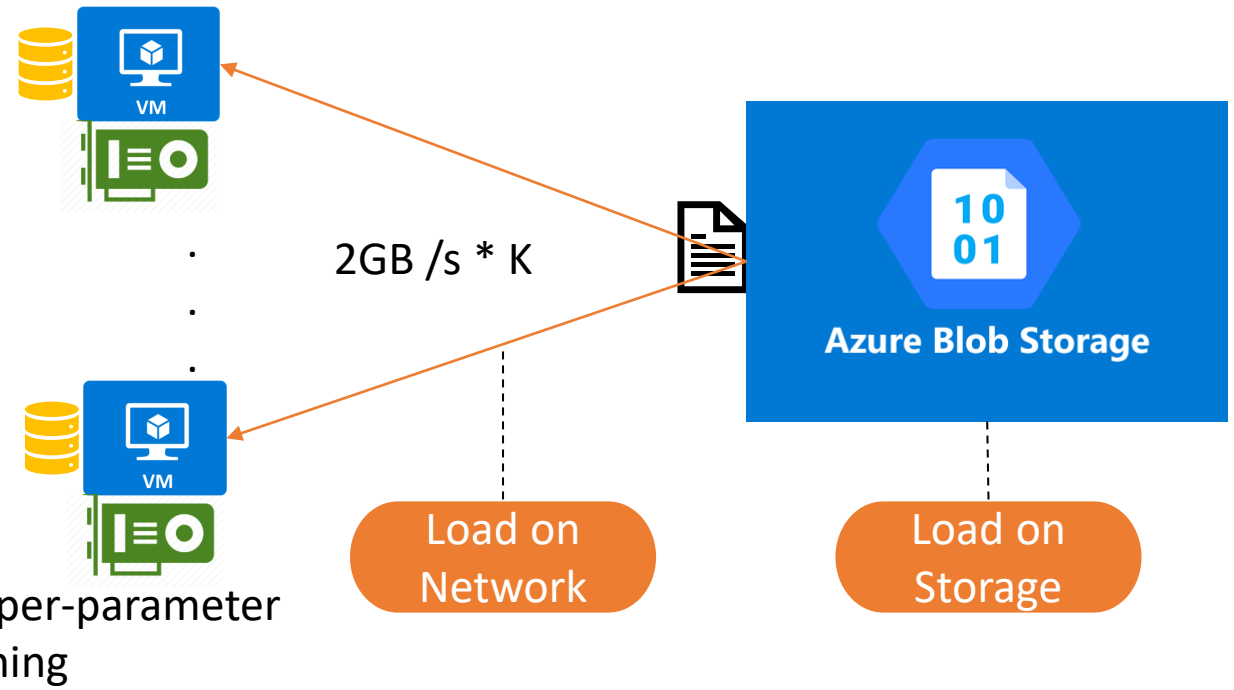
# Example workload

- Resnet50 is a popular vision model
- Process 10,500 images/sec on 8 Nvidia V100s
- **Goal: Keep GPUs busy and utilize them efficiently**



# Example workload

- Resnet50 is a popular vision model
- Process 10,500 images/sec on 8 Nvidia V100s
- **Goal: Keep GPUs busy and utilize them efficiently**




- Cheap Preemptible VMs => Job Migration
- Large datasets

Remote store with several TBs of training data

# Quiver: Key ideas

- Domain specific intelligence at caching layer
  - Substitutability – Use existing contents of the cache to avoid thrashing
- Hash-based content addressing for security
- Co-designed with deep-learning framework (PyTorch)
- Dynamically manages cache allocation
- Improve cluster throughput up-to 2.3x

# Structure

- Introduction & Motivation
- Background
- Design
- Implementation
- Evaluation

# Background: Deep Learning

- Learn a model to **represent** training data
- Iterate over random subsets of input data – **Mini batch**
- Perform **Gradient Descent (SGD)** on each mini-batch
- Process the entire dataset in random order – **Epoch**


# A cache for DLT jobs

- DLT datasets are accessed multiple times
  - Within same job:  Multiple epochs read the entire dataset
  - Across jobs:  Hyperparameter exploration, popular datasets (e.g. ImageNet)
- Good fit for caching

# A cache for DLT jobs

- DLT datasets are accessed multiple times
  - Within same job:  Multiple epochs read the entire dataset
  - Across jobs:  Hyperparameter exploration, popular datasets (e.g. ImageNet)
- Good fit for caching

- Challenges
  - **Random access** within epoch => Partial caching can cause thrashing (e.g. LRU)
  - **Job Heterogeneity** => Not all jobs benefit the same from caching
  - **Secure inter-job data access**

# A cache for DLT jobs

- DLT datasets are accessed multiple times
  - Within same job:  Multiple epochs read the entire dataset
  - Across jobs:  Hyperparameter exploration, popular datasets (e.g. ImageNet)
- Good fit for caching

- Challenges
  - **Random access** within epoch => Partial caching can cause thrashing (e.g. LRU)
  - **Job Heterogeneity** => Not all jobs benefit the same from caching
  - **Secure inter-job data access**
- **Quiver: Use domain intelligence to address these challenges**

# #1: Thrashing-proof partial caching

- Two I/O properties
  - Each input touched once in an epoch
  - Every mini-batch needs to be randomly sampled

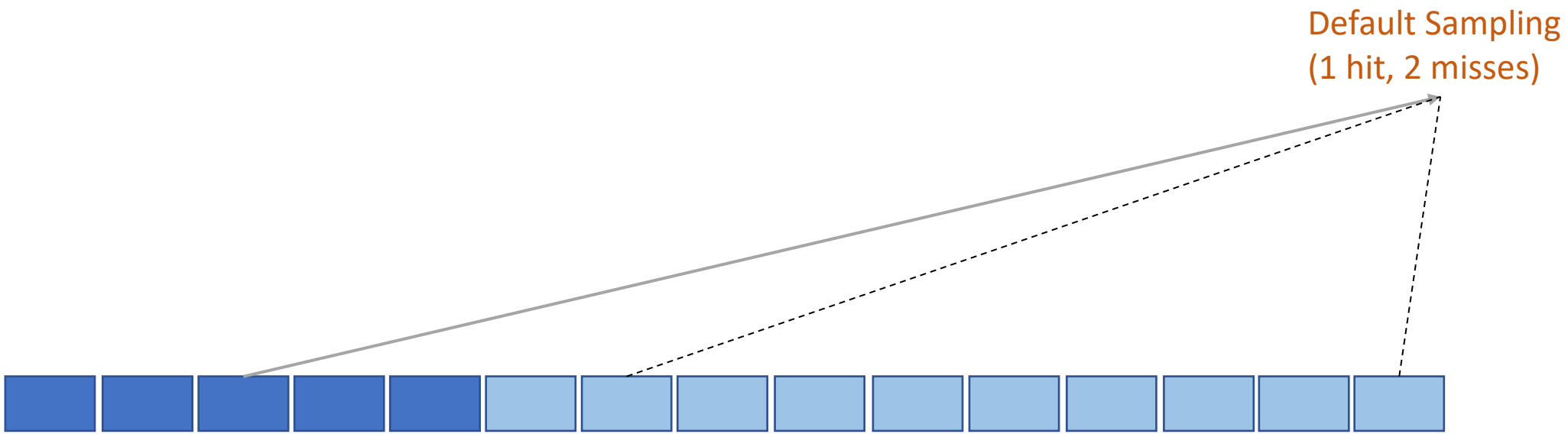
- **Substitutable hits**
  - I/O is substitutable
  - Mini-batch samples order does not matter, as long as it is random

# #1: Thrashing-proof partial caching

- Substitutability while sampling
- Looks up more than the number of indices and returns whatever is in the cache (*substitutable hits*)

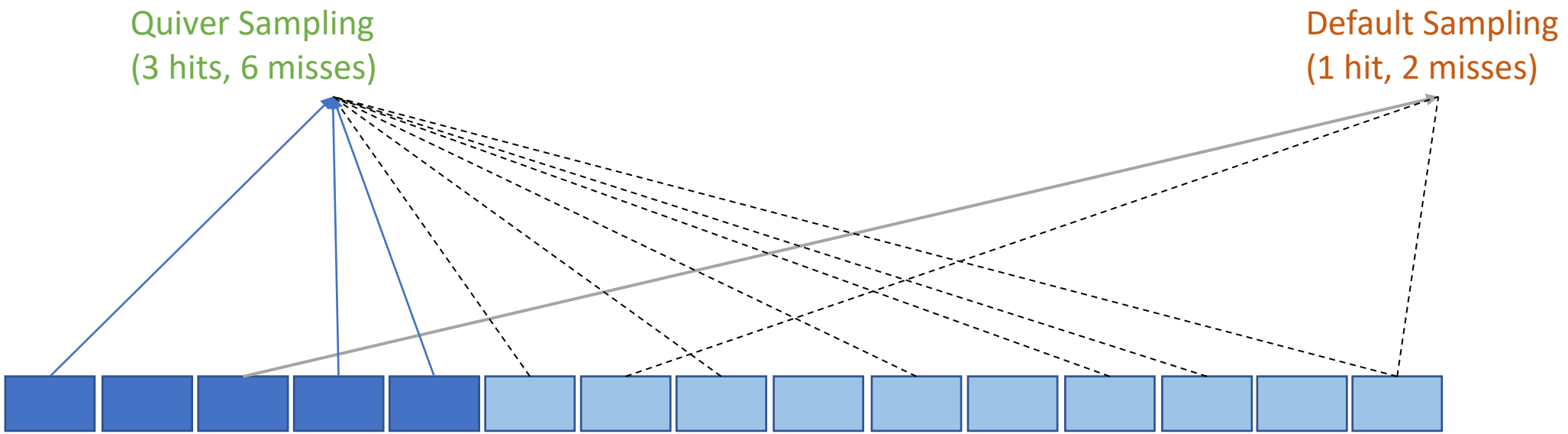
# #1: Thrashing-proof partial caching

- Substitutability while sampling
- Looks up more than the number of indices and returns whatever is in the cache (*substitutable hits*)



# #1: Thrashing-proof partial caching

- Substitutability while sampling
- Looks up more than the number of indices and returns whatever is in the cache (*substitutable hits*)



# #2: Job heterogeneity and caching

- Benefit-aware caching to handle Job heterogeneity
  - Time per mini-batch is an application-specific metric for performance
    - Allows cheap profiling to measure benefits from cache
- **Predictability**
  - Measure time per minibatch with different caching modes
  - Given total space budget, the manager allocates cache per dataset

# #3: Secure Inter-Job Data access

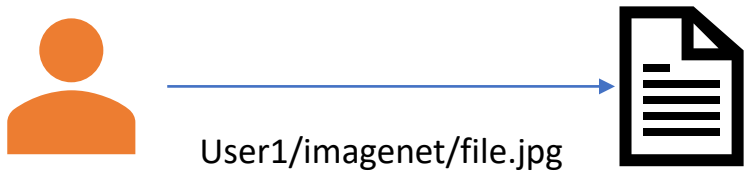
- Multiple jobs and users share cache
- Data needs reuse/sharing while retaining isolation
- Each file is addressed by its hash instead of its name

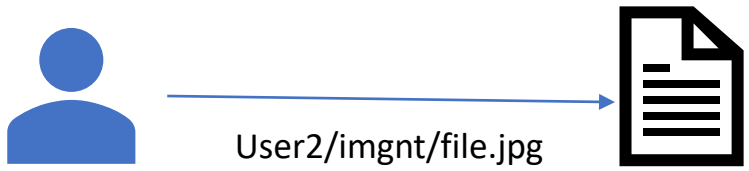
# #3: Secure Inter-Job Data access

- Multiple jobs and users share cache
- Data needs reuse/sharing while retaining isolation
- Each file is addressed by its hash instead of its name

# #3: Secure Inter-Job Data access

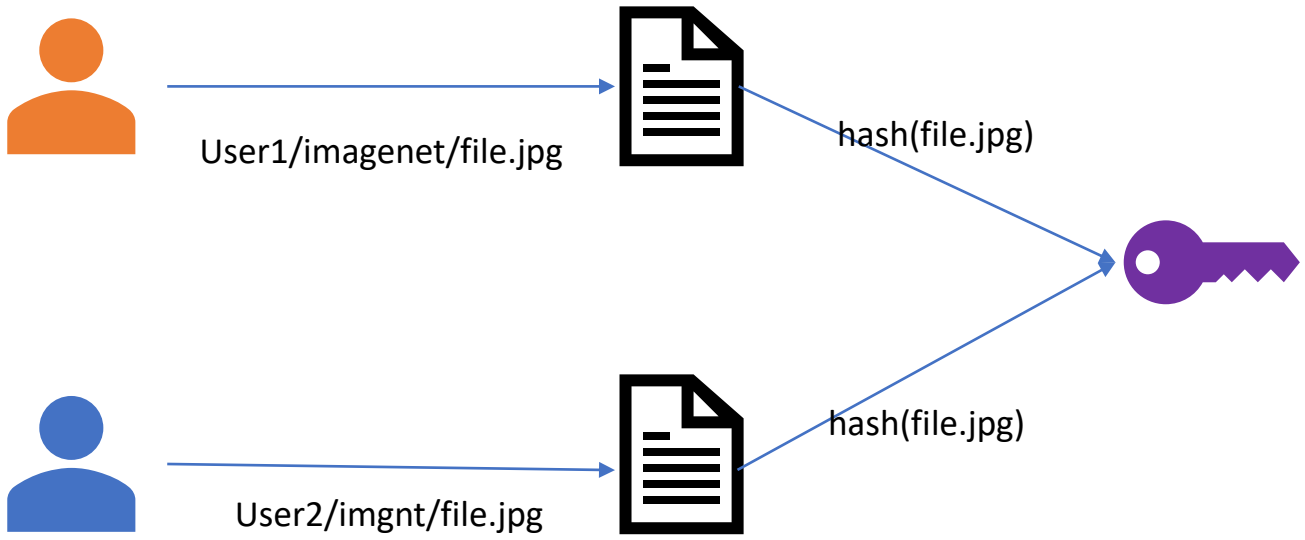
- Multiple jobs and users share cache
- Data needs reuse/sharing while retaining isolation
- Each file is addressed by its hash instead of its name



# #3: Secure Inter-Job Data access

- Multiple jobs and users share cache
- Data needs reuse/sharing while retaining isolation
- Each file is addressed by its hash instead of its name

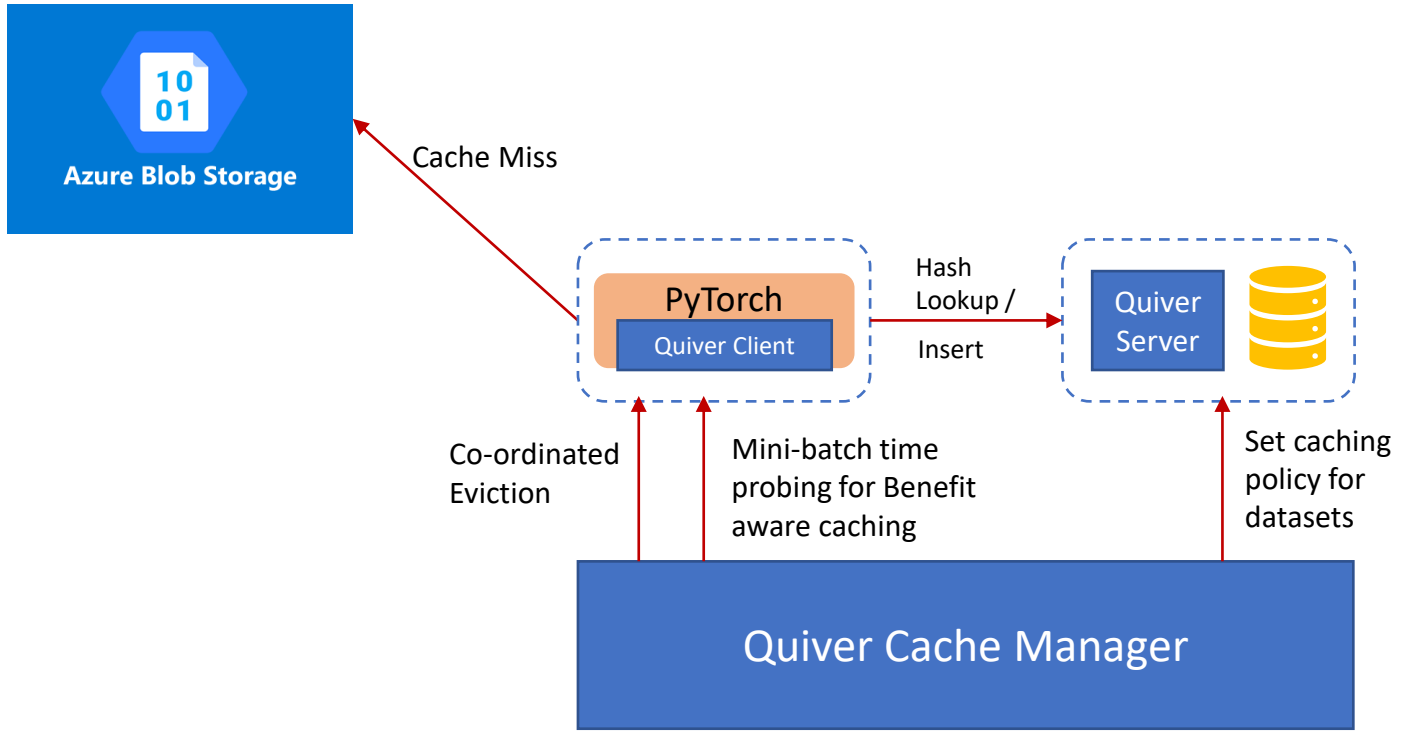


# Structure

- Introduction & Motivation
- Background
- Design
- Implementation
- Evaluation

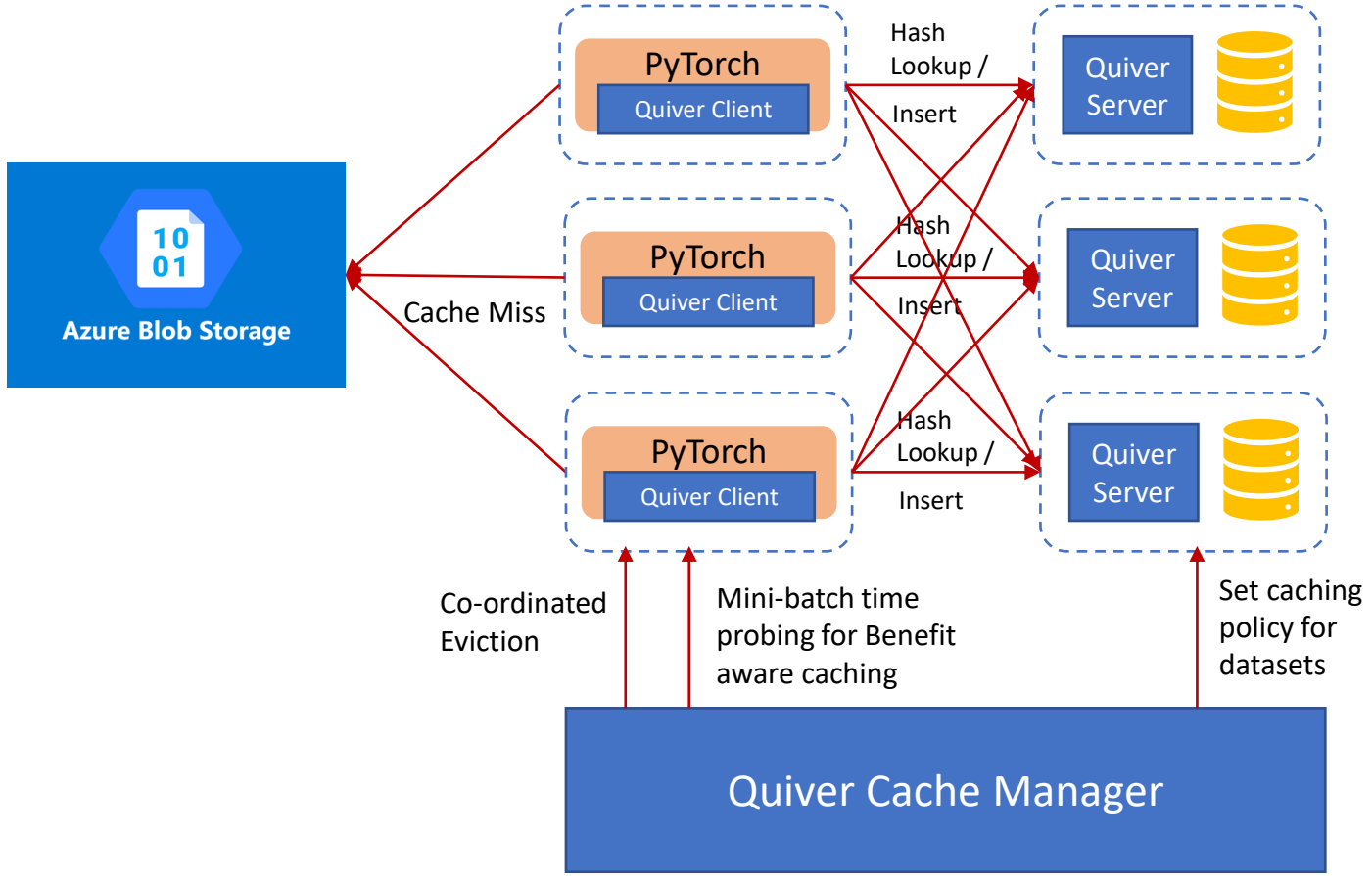
# Architecture of Quiver

- Quiver cache server
- Quiver cache client co-designed with PyTorch
- Quiver cache manager
- Quiver instance types
  1. Entire cluster
  2. Each rack



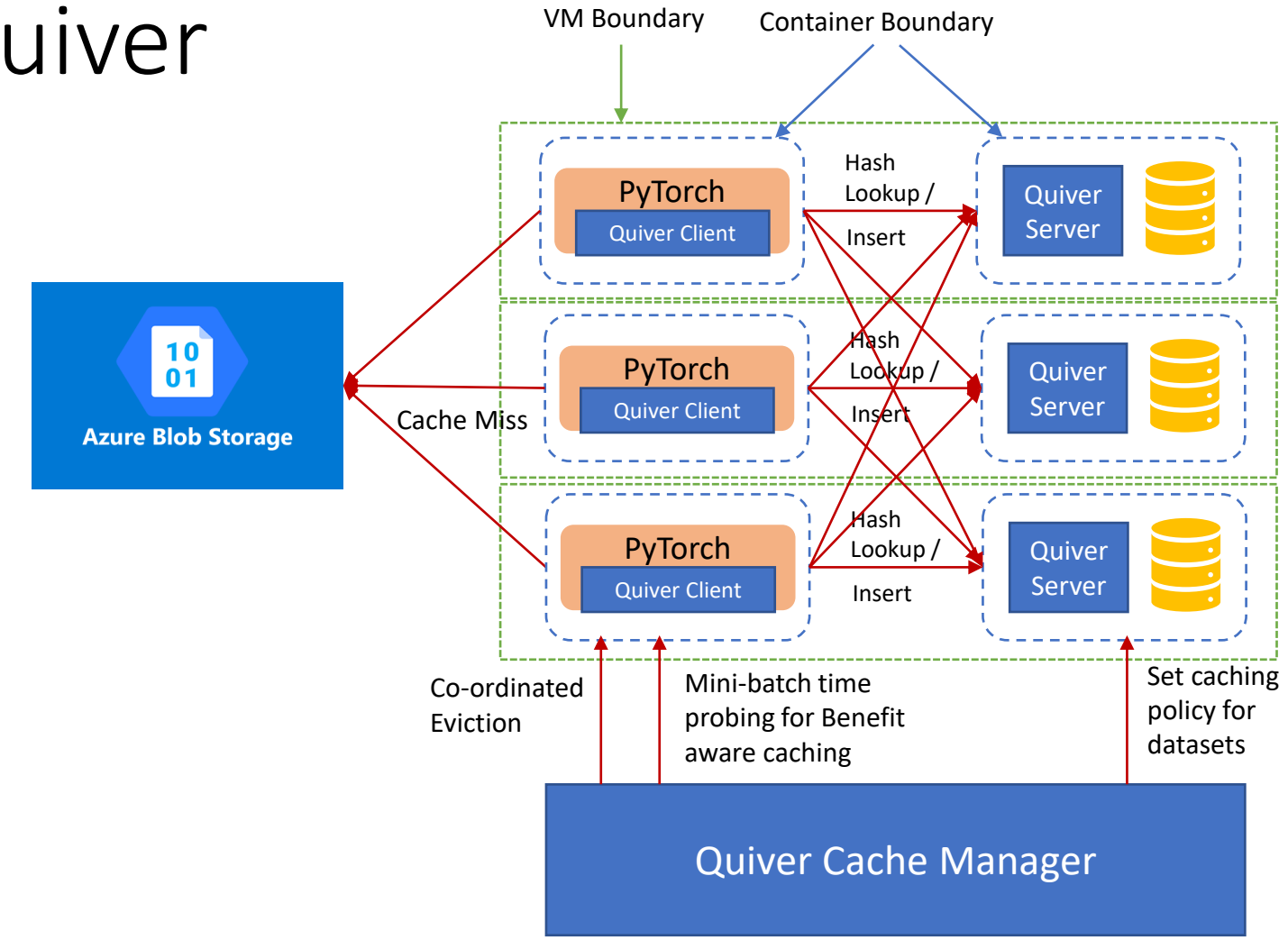
# Architecture of Quiver

- Quiver cache server
- Quiver cache client co-designed with PyTorch
- Quiver cache manager
- Quiver instance types
  1. Entire cluster
  2. Each rack



# Architecture of Quiver

- Quiver cache server
- Quiver cache client co-designed with PyTorch
- Quiver cache manager
- Quiver instance types
  1. Entire cluster
  2. Each rack



# Cache Access

- Client is integrated with PyTorch data-layer
  - Fetches files from remote on misses
  - Populates the cache servers
- Works with **hash-digest** file
- Incorporates ***substitutable hits*** and ***co-operative*** **miss handling**

# Hash digest and Partition

- Dataset is represented by a hash-digest
- Major components of an entry in the hash-file
  - <content_hash: file_location>
- Key space is partitioned across servers

# Hash digest and Partition

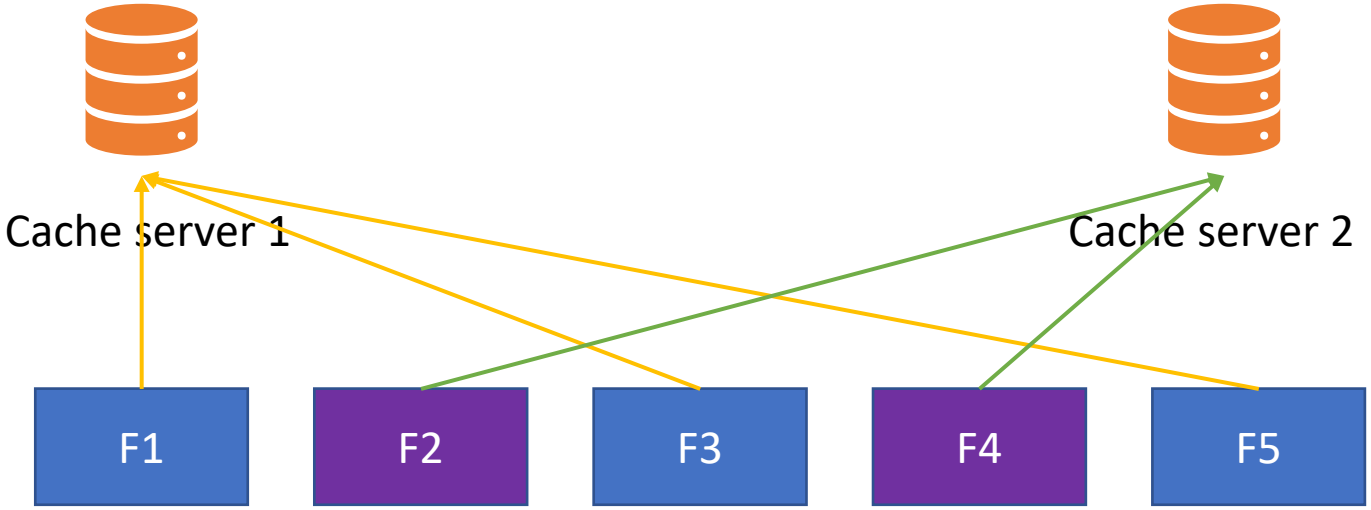
- Dataset is represented by a hash-digest
- Major components of an entry in the hash-file
  - <content_hash: file_location>
- Key space is partitioned across servers

Cache server 1

Cache server 2

# Hash digest and Partition

- Dataset is represented by a hash-digest
- Major components of an entry in the hash-file
  - <content_hash: file_location>
- Key space is partitioned across servers



# Co-operative miss handling

- Misses are sharded across jobs using same dataset.
  - Sharding is implicit by randomizing indices
  - *Happens naturally in DLT access pattern*
  - *Jobs benefit from other jobs as they progress*

# Co-operative miss handling

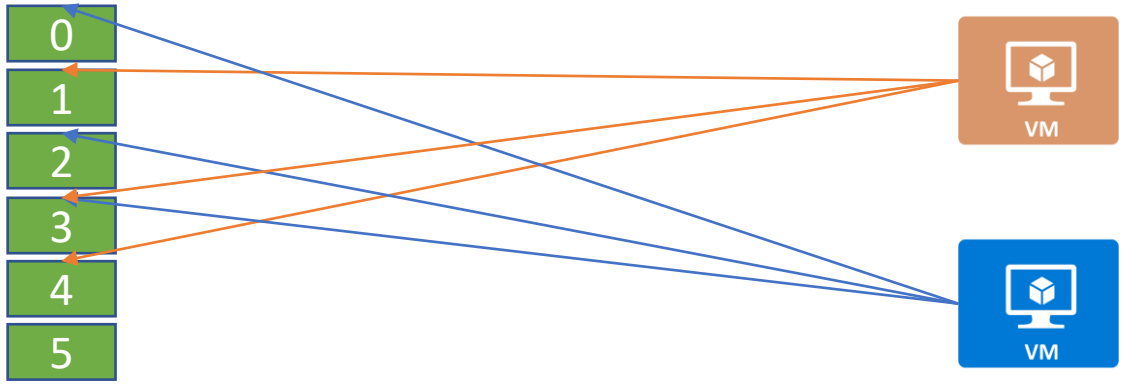
- Misses are sharded across jobs using same dataset.
  - Sharding is implicit by randomizing indices
  - *Happens naturally in DLT access pattern*
  - *Jobs benefit from other jobs as they progress*


# Co-operative miss handling

- Misses are sharded across jobs using same dataset.
  - Sharding is implicit by randomizing indices
  - *Happens naturally in DLT access pattern*
  - *Jobs benefit from other jobs as they progress*

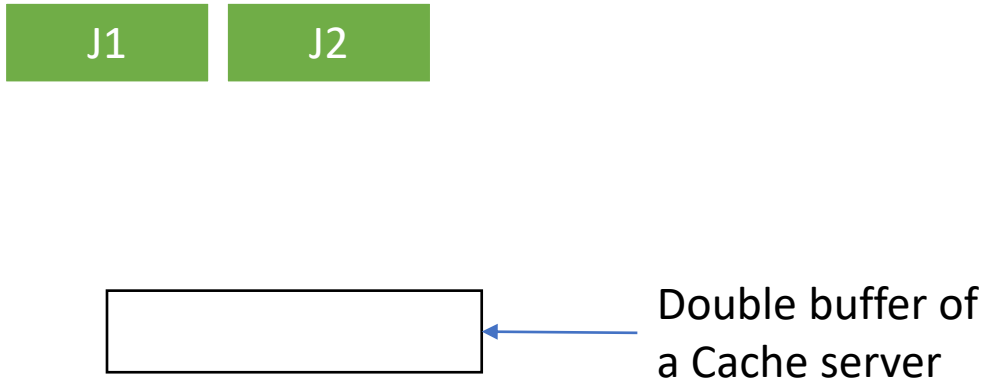
# Co-operative miss handling

- Misses are sharded across jobs using same dataset.
  - Sharding is implicit by randomizing indices
  - *Happens naturally in DLT access pattern*
  - *Jobs benefit from other jobs as they progress*



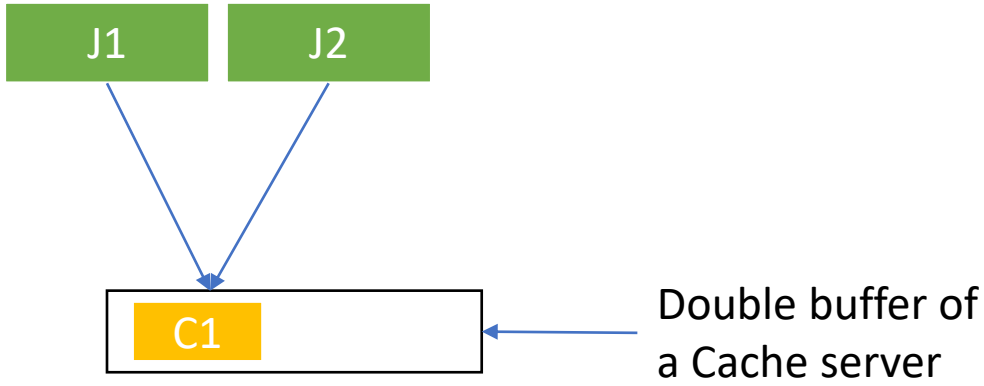
# Co-ordinated eviction

- Dataset partition
  - Digest file is partitioned into given number of chunks
- Double buffering of chunks
  - Chunks allow coordinated access of cache
  - Co-ordinated eviction
    - Mark for eviction – no new refs
    - Then evict
    - Similar to UNIX **unlink** call



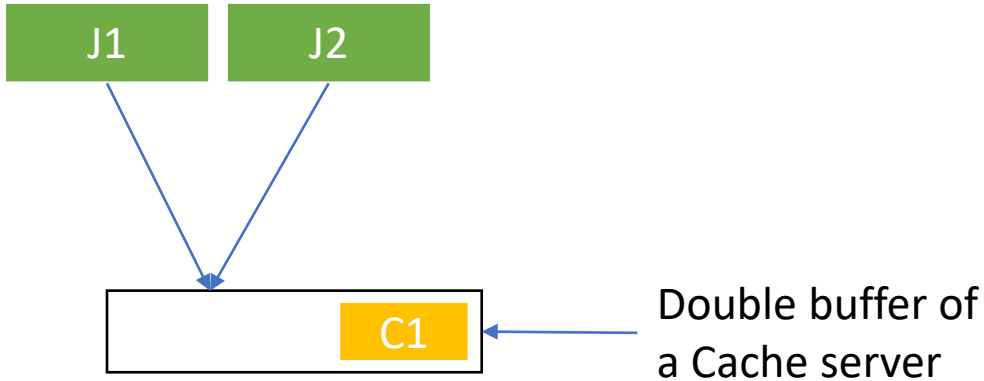
# Co-ordinated eviction

- Dataset partition
  - Digest file is partitioned into given number of chunks
- Double buffering of chunks
  - Chunks allow coordinated access of cache
  - Co-ordinated eviction
    - Mark for eviction – no new refs
    - Then evict
    - Similar to UNIX **unlink** call



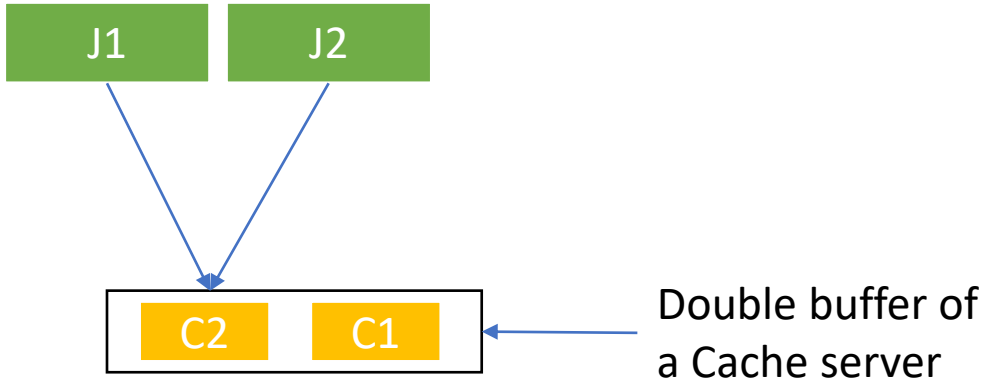
# Co-ordinated eviction

- Dataset partition
  - Digest file is partitioned into given number of chunks
- Double buffering of chunks
  - Chunks allow coordinated access of cache
  - Co-ordinated eviction
    - Mark for eviction – no new refs
    - Then evict
    - Similar to UNIX **unlink** call



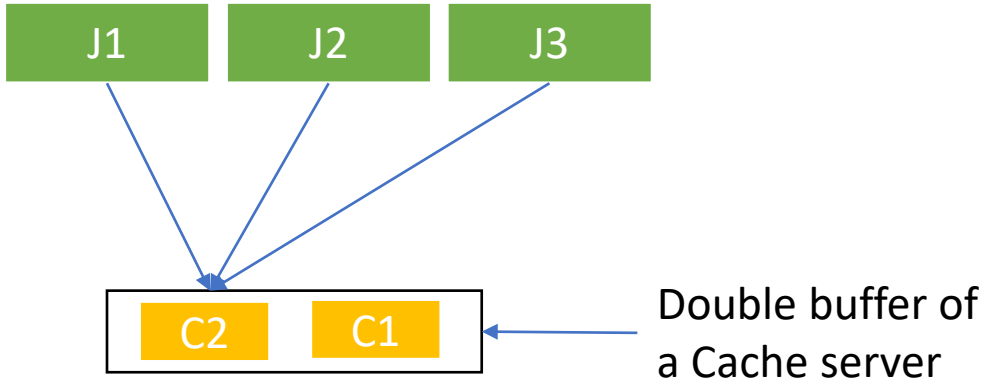
# Co-ordinated eviction

- Dataset partition
  - Digest file is partitioned into given number of chunks
- Double buffering of chunks
  - Chunks allow coordinated access of cache
  - Co-ordinated eviction
    - Mark for eviction – no new refs
    - Then evict
    - Similar to UNIX **unlink** call



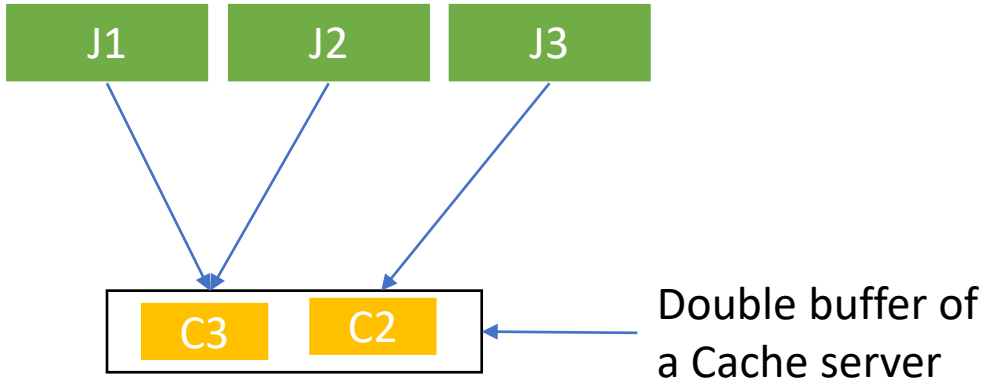
# Co-ordinated eviction

- Dataset partition
  - Digest file is partitioned into given number of chunks
- Double buffering of chunks
  - Chunks allow coordinated access of cache
  - Co-ordinated eviction
    - Mark for eviction – no new refs
    - Then evict
    - Similar to UNIX **unlink** call



# Co-ordinated eviction

- Dataset partition
  - Digest file is partitioned into given number of chunks
- Double buffering of chunks
  - Chunks allow coordinated access of cache
  - Co-ordinated eviction
    - Mark for eviction – no new refs
    - Then evict
    - Similar to UNIX **unlink** call



# Structure

- Introduction & Motivation
- Design
- Implementation & Evaluation

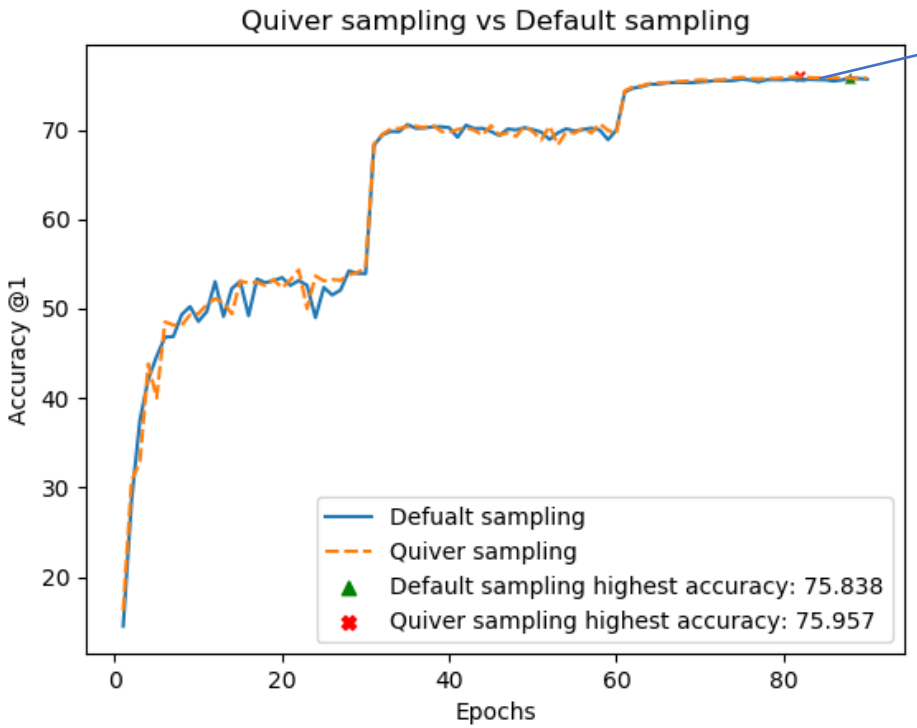
# Implementation

- Cache client (900 LoC)
  - Dataloader of PyTorch (v 1.1.0)
  - Dataset of PyTorch
  - Sampler of PyTorch
- Cache server (1200 LOC)
  - A C++ key value store
- Cache manager
  - A python program


# Evaluation Setup

- Cluster (48 GPUs)
  - 6 VMs with 4 NVIDIA P100 GPUs
  - 6 VMs with 4 NVIDIA P40 GPUs
- Workloads
  - Resnet50 on Imagenet dataset (154 GB)
  - Inception_V3 on openimages dataset (531 GB)
  - DeepSpeech2 on LibriSpeech dataset (90 GB)

# Impact on accuracy



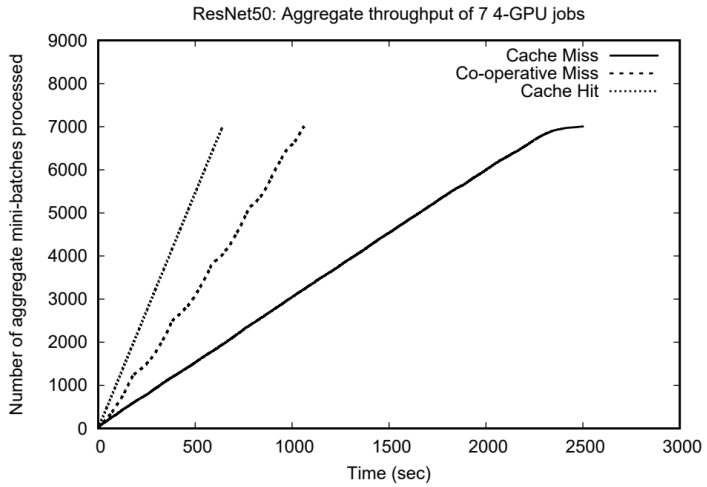

RESNET50 on Imagenet

Similar curves

| Config | Word Error Rate (WER) |
|---|---|
| Baseline Sampling | 22.29 |
| Quiver Sampling | 22.32 |

DeepSpeech2 on LibriSpeech

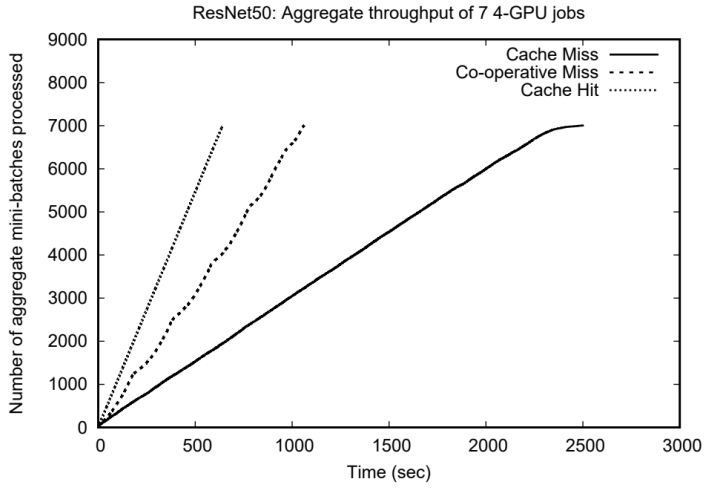
# Throughput increase because of quvier



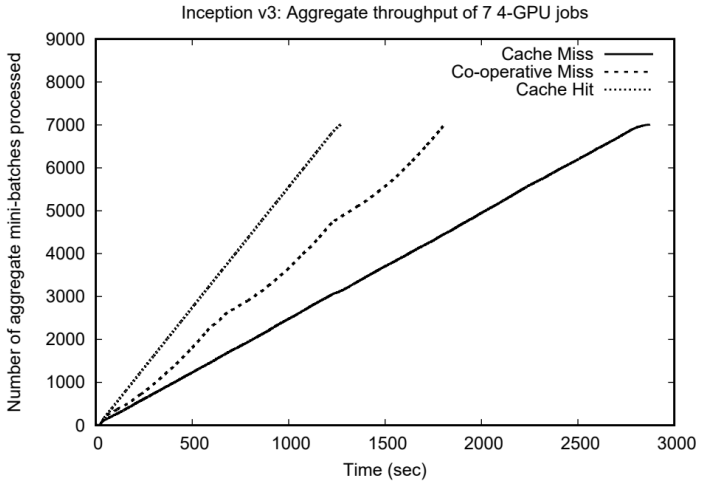

Resnet50

| Workload | Time for 7000 mini-batches (s) | | |
|---|---|---|---|
| | Baseline | HIT | CO-OP |
| Resnet50 | 2505 | 646 (3.88x) | 1064 (2.35x) |

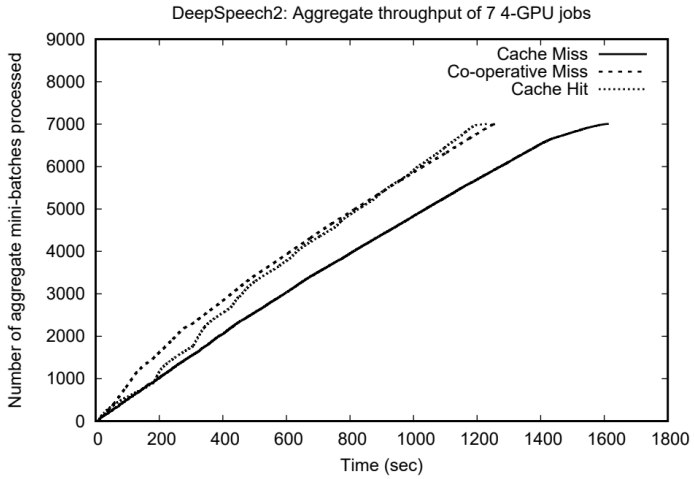
# Throughput increase because of quvier




Resnet50



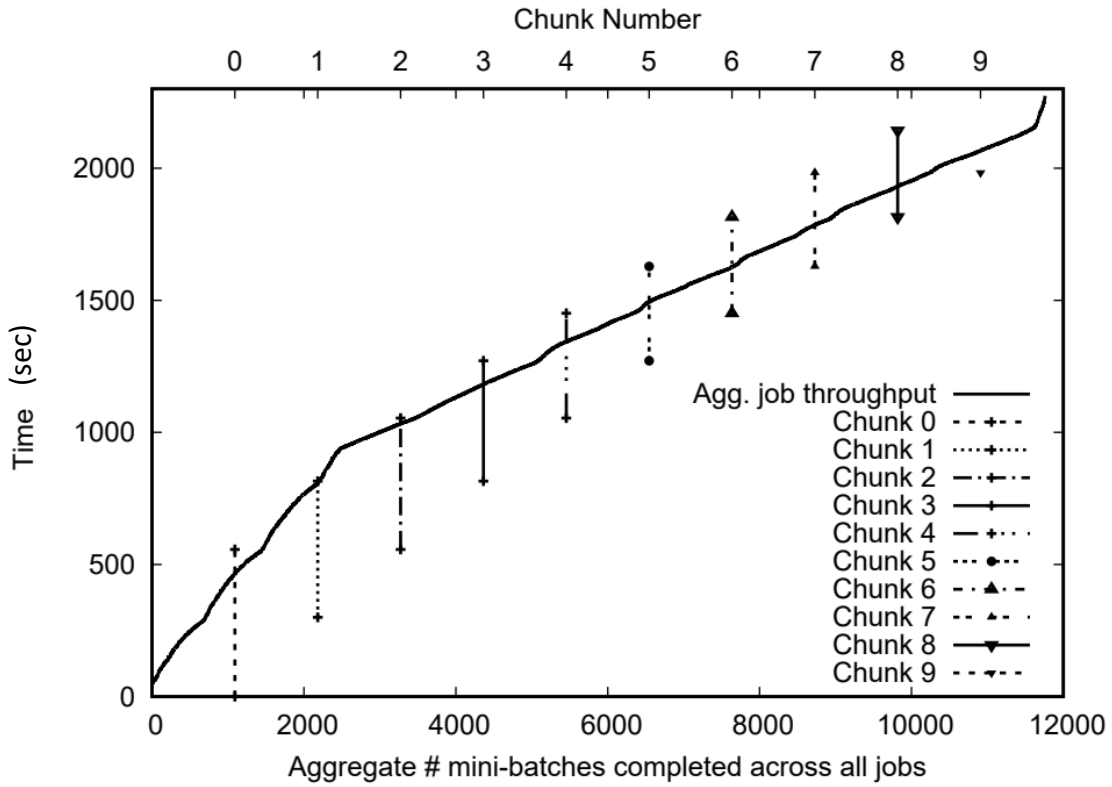

InceptionV3



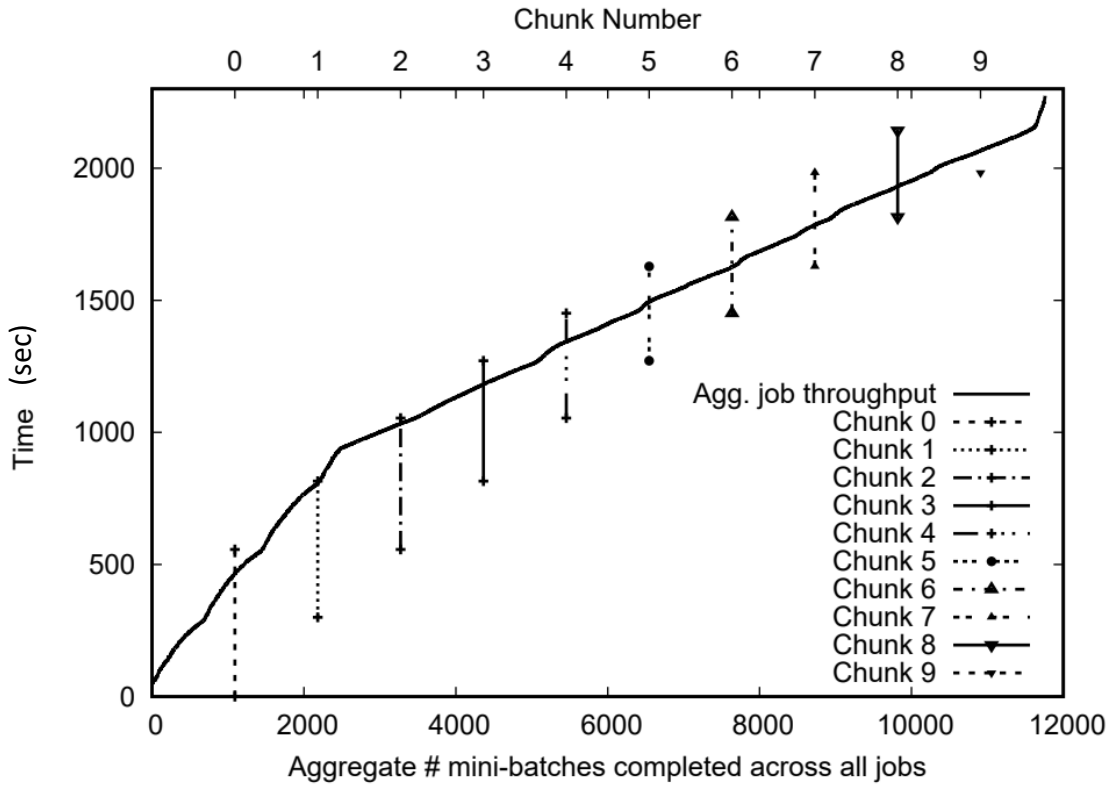

DeepSpeech2

| Workload | Time for 7000 mini-batches (s) | | |
|---|---|---|---|
| | Baseline | HIT | CO-OP |
| Resnet50 | 2505 | 646 (3.88x) | 1064 (2.35x) |
| Inception | 2874 | 1274 (2.26x) | 1817 (1.58x) |
| DeepSpeech | 1614 | 1234 (1.31x) | 1265 (1.28x) |

# Co-ordinated eviction in action

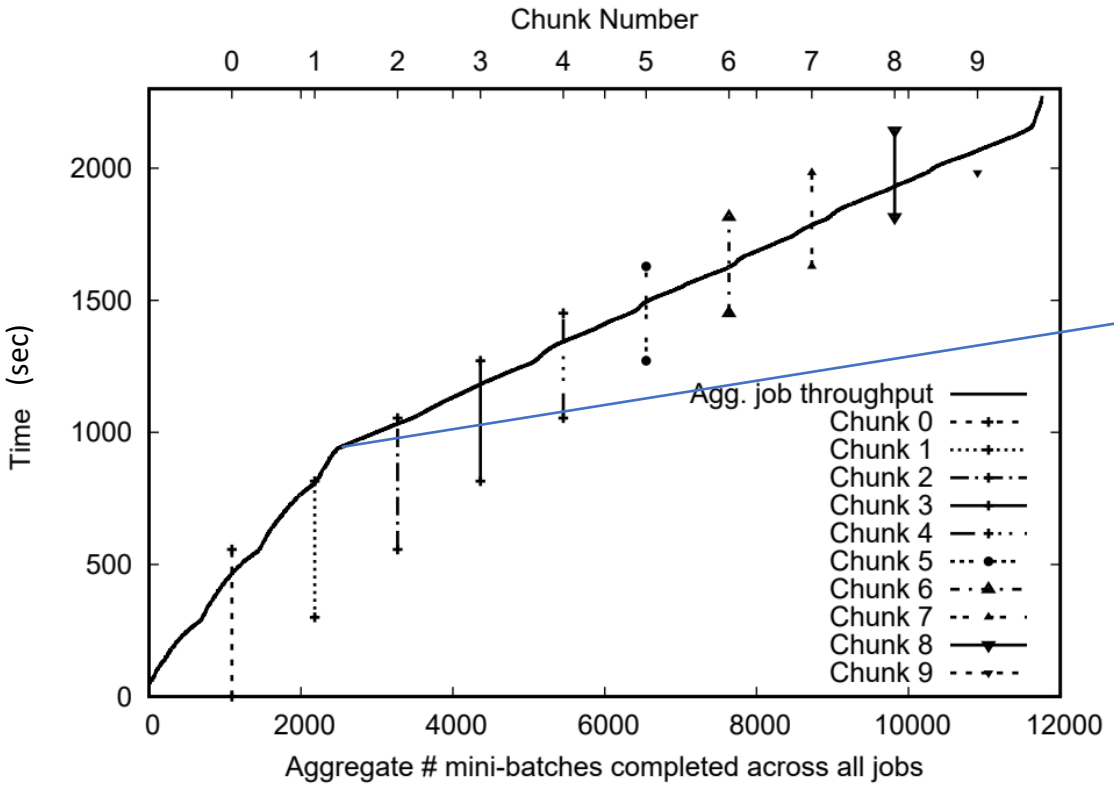


# Co-ordinated eviction in action



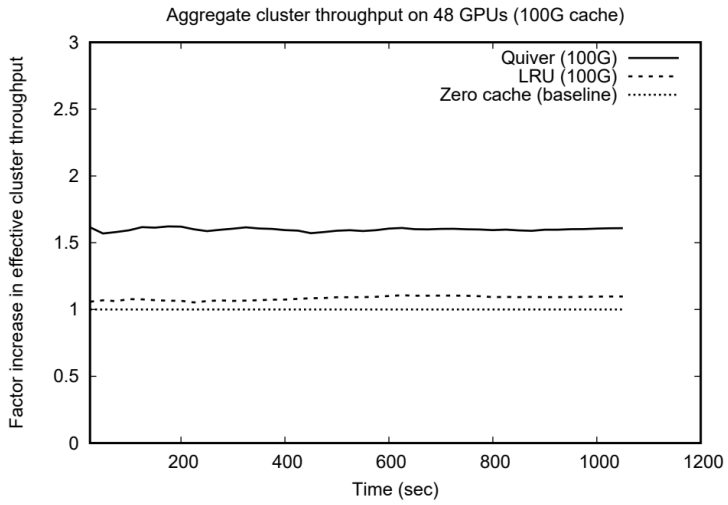

- 2 Chunks cached at a time
- New jobs start using 3rd chunk

# Co-ordinated eviction in action

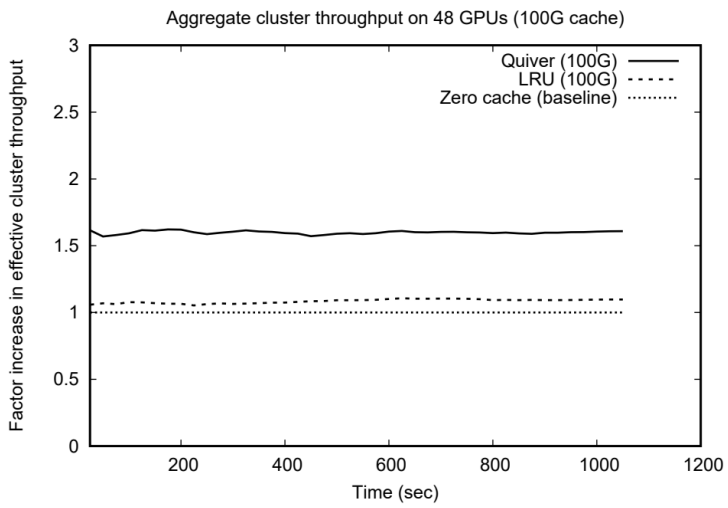


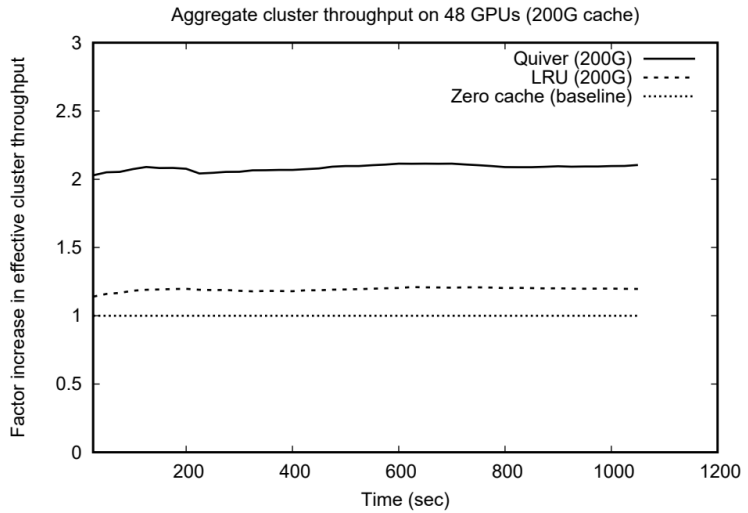

- 2 Chunks cached at a time
- New jobs start using 3rd chunk

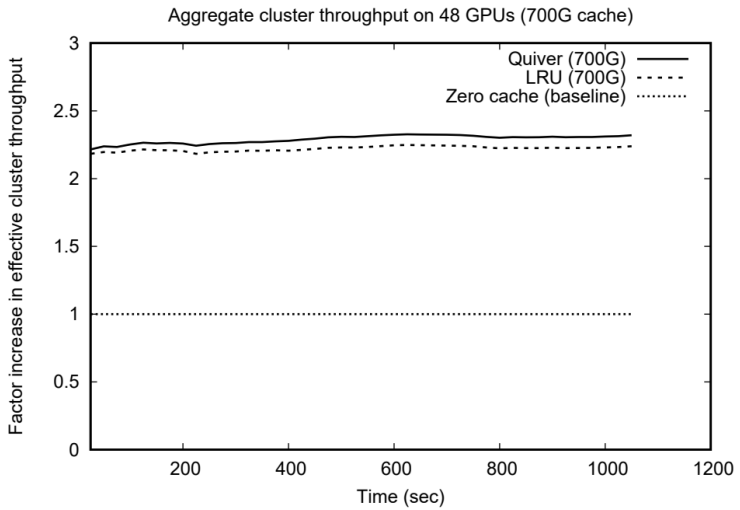
# Benefit aware caching



# Benefit aware caching




- Mixed workload – 12 Different jobs
- Quiver preferentially allocates cache to different datasets
- Quiver yields sizeable benefits even with tiny cache (100G)
- Improvement in cluster throughput ranges between 1.6x to 2.3x

# Summary

- Quiver is a domain-specific storage cache for DLT jobs
- Utilizes I/O behavior of deep learning training jobs
  - Substitutable hits => New thrash-proof partial caching
  - Predictability => Benefit-aware caching
- Improves cluster GPU utilization by reducing I/O wait time
- Implemented in PyTorch