Campo: Cost-Aware Performance Optimization for Mixed-Precision Neural Network Training

Cachew: Machine Learning Input Data Processing as a Service

Infrastructure Software Lab

CAMPO

Overview

- A tool that improves performance of mixed-precision NN training with awareness of casting costs
- Campo assigns the low or full computation precision to each operation in NN to maximize performance while preserving the NN model accuracy
- Campo is built upon performance modeling that predicts the casting cost and operation performance with low precision, and introduces a cost-aware graph rewriting strategy
- This strategy avoids applying low precision to those operations that cannot get performance benefit, and minimizes the casting cost when applying low precision to a group of operations
- In addition, some operations can benefit from low precision, but cannot run on TC because their input data sizes cannot meet the requirement of TC. For those operations, Campo pads the input tensors without programmer participation to maximize the utilization of TC for high performance
- Only considers FP16 as lower precision (similarly to TF and PyTorch), even if INT8 and INT4 are available

Mixed Precision Training

The mixed precision graph optimizer decides the assignment of low precision to operations by classifying operations into multiple lists based on operation's numerical safety (numerical safety refers to how an NN model's quality is affected by the use of low precision):

Allowlist

- Numerically-safe execution in FP16, is also performance-critical → Always FP16 (e.g. MatMul, Conv2D)

Denylist

- Numerically-dangerous in FP16 & their effects may also be observed in a downstream operation → Always FP32 (e.g. Exp, SoftMax)

Inferlist

- Numerically-safe in FP16, but may be made unsafe by an upstream denylist operation (e.g. BiasAdd)

Clearlist

- No numerically-significant effects (FP16 and FP32) (e.g. Max and Min)

Using the above lists, the mixed precision graph optimizer uses low precision for an operation, if any of the following three conditions is true:

- (1) The operation is in the allowlist;
- (2) the operation is in the clearlist, and its immediate ancestor(s) and immediate descendent(s) are using low precision;
- (3) the operation is in the inferlist and there is no upstream denylist operation. The mixed precision graph optimization works online by re-writing the dataflow graph via inserting casts before the iterative training of NN model happens

Tensor Core Acceleration / Observations

- Compared to regular CUDA cores, TC is more performant and energy-efficient. TC is used to accelerate FP16 matrix multiplication and convolution operations
- TC is automatically activated to run an operation when two conditions are met:
  - (1) the operation is either matrix multiplication or convolution using FP16
  - (2) the input tensors of the operation satisfy the shape requirements

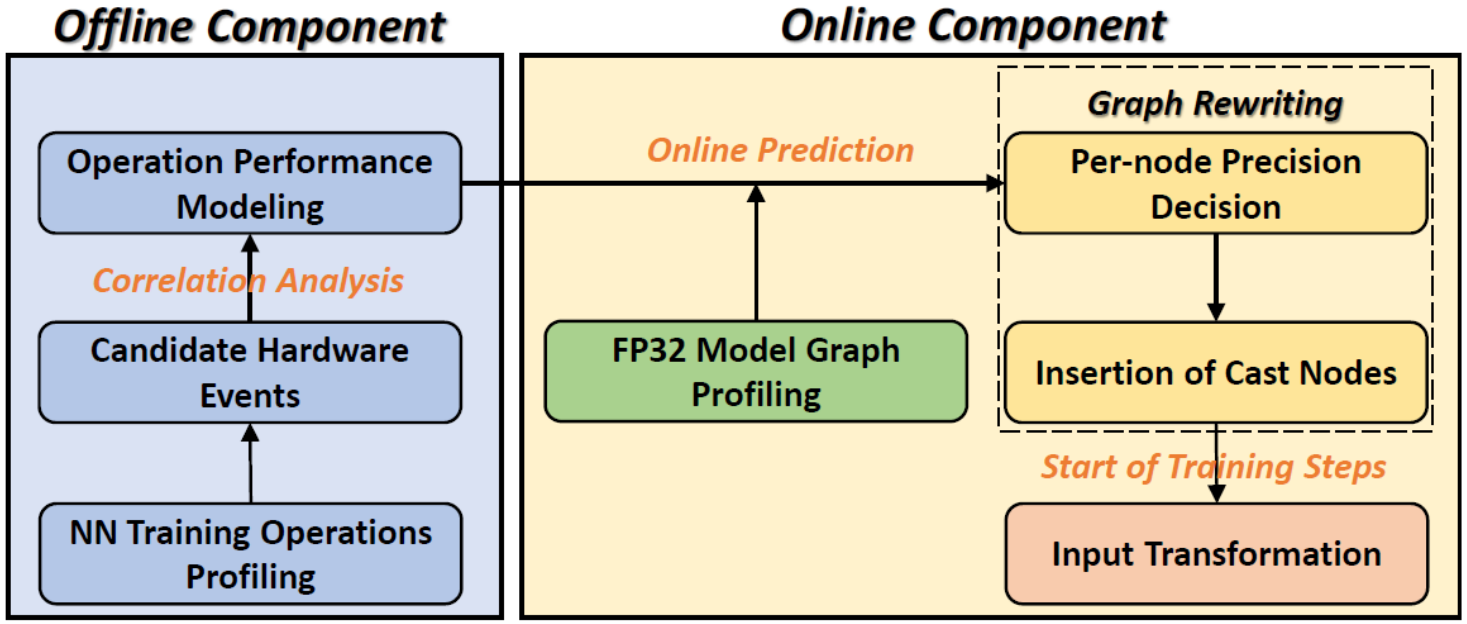
Observation #1: Without TC, using F16 leads to slightly better performance than using F32. Using TC for FP16 magnifies the performance benefit of F16;

Observation #2: The performance gain of using FP16 varies largely across input data sizes;

Observation #3: The cast operation introduces non-negligible overhead. Considering the casting cost, it is not always performance-profitable to convert FP32 to FP16 regardless of using TC or not;

Implications of observations: The effectiveness of using low precision for an operation is impacted by input data size, casting cost, and the usage of TC. Optimizing the assignment of low precision to operations is a multi-dimensional problem, not just one-dimensional problem as assumed in the existing solutions.

System Overview



- The **offline component** is used to **build performance modeling to predict performance of operations in FP16**;
- The **performance modeling**:
  - is **used by the online component**, and **makes performance prediction using performance events** (e.g., L2 cache misses and global memory load/store throughput) collected from operation execution in FP32;
  - **uses correlation analysis to decide which events are the most important for accurate performance prediction;**
  - **is based on statistical regression modeling**, which is built only once by the offline component but repeatedly used by the online component;
- **The online component includes graph profiling, graph traverse to make per-node precision decision, insertion of cast operation, and input transformation;**
- The **graph profiling** uses one iteration to run operation nodes in FP32 and collect events needed by performance modeling;
- The **graph traverse** makes four passes on the dataflow graph of the NN model and uses performance modeling to decide if each operation should use FP16 or not based on the estimation of performance benefit and cost of using FP16;

Performance modelling: methodology / casting cost

- Predict casting cost and operator's performance in FP16
- Use lightweight regression-based modelling
- Operation-specific performance modelling

CASTING COST

- Casting cost is related to input tensor size and CastOp instance initialization time (C_I)

$$casting\_cost = r * tensor\_size + C_I$$

- **Predict low-precision performance:**
  - Specific to each operation
  - Performance-critical hardware events as model features
  - Spearman's rank correlation analysis for feature selection

$$OP_{LP} = OP_{FP32} \cdot (\sum_{i=1}^{N} w_i \cdot PC_i) + \sigma$$

Runtime graph rewriting

Task at Runtime:

- Decide data precision for each operation;
- Decide which operations to be converted together to reduce the number of cast operation nodes;

Goals:

- Minimizing the training time;
- Minimizing the casting cost;
- No adverse impact on the numerical safety (compared with the traditional mixed-precision training)

4 graph traverses

- **Traverse 1:**
  - Are the nodes in *allowlist* (i.e. numerical-safe nodes)?
  - The performance benefit is larger than the casting cost?
- **Traverse 2:**
  - Check numerically-unsafe nodes → No hope for conversion
- **Traverse 3:**
  - Check if remaining nodes are in inferlist or clearlist → Determine their numerical-safety if converting
- **Traverse 4:**
  - For those numerically-safe nodes identified in the last traverse, the performance benefit is larger than the casting cost?

Experimental setup

Experimental platform

- CPU: Intel Xeon CPU E52648L v4@ 1.80GHz
- GPU: (1) Nvidia RTX 2080 Ti and (2) V100

Workloads

- Alexnet, Inception3, Vgg16, Resnet50, DCGAN, BERT-Large

Datasets

- ImageNet, CelebA, SQuAD

Baseline

- The state-of-the-art AMP in TensorFlow (TF_AMP)
- Full-precision training (FP32)

Training Throughput

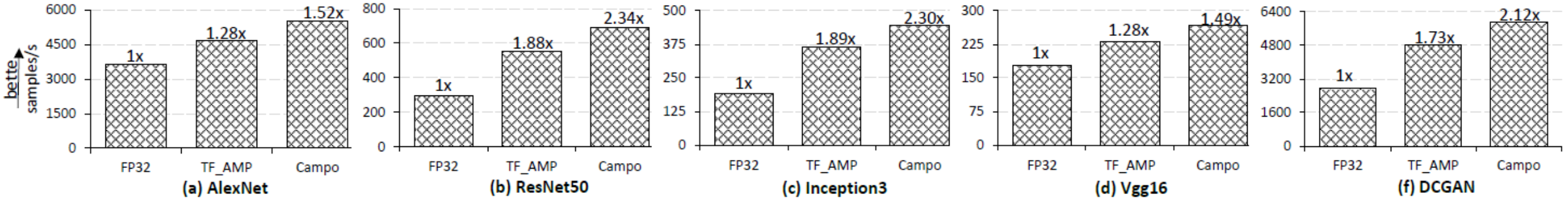


Figure 4: Training throughput with FP32, TF_AMP and Campo on RTX 2080 Ti.

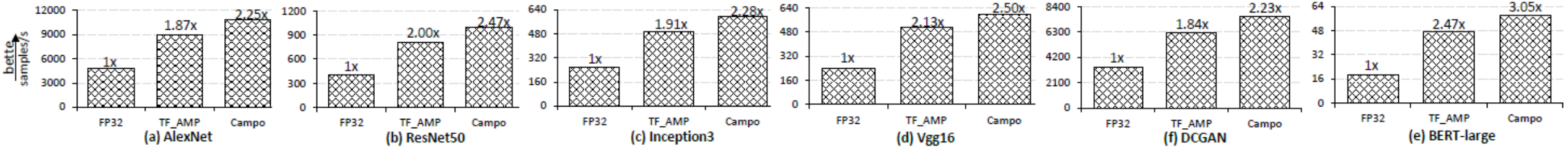
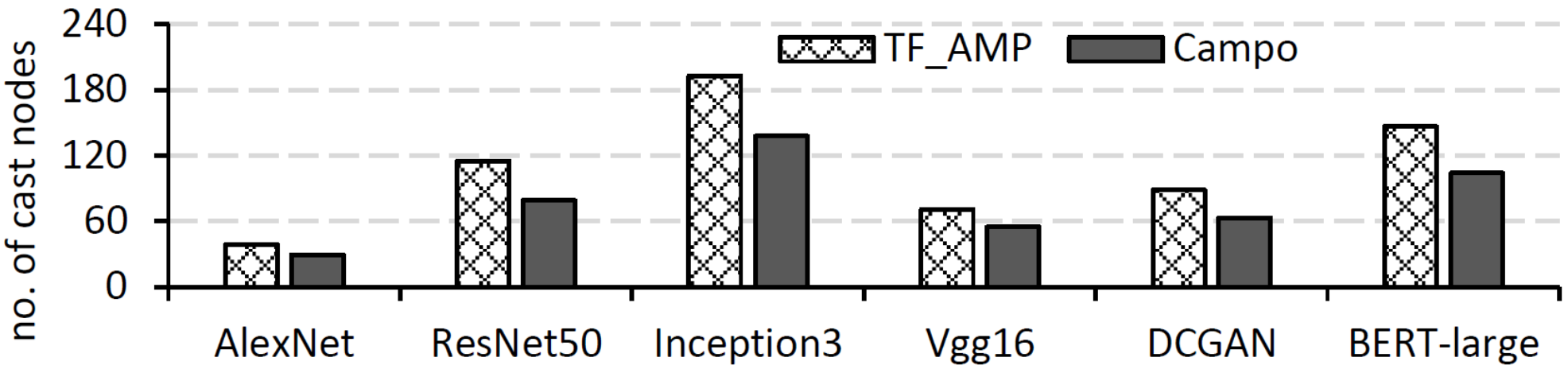


Figure 5: Training throughput with FP32, TF_AMP and Campo on V100.

Campo improves throughput by 20.9% on average (up to 23.4%) compared to TF_AMP

FUTUREWEI Technologies

Number of cast operation nodes



Number of cast nodes of NN models trained with TF_AMP and Campo, respectively

Campo uses 27.7% less cast operation nodes on average (up to 31.3%) than TF_AMP

FUTUREWEI Technologies

TC utilization

NN models	Top-1 Accuracy (%)			Top-5 Accuracy (%)		
	FP32	TF_AMP	Campo	FP32	TF_AMP	Campo
AlexNet	63.39	64.41	64.38	81.24	81.21	81.19
ResNet50	78.77	78.74	78.75	94.86	94.82	94.85
Inception3	78.42	78.45	78.43	90.15	90.16	90.15
Vgg16	71.58	71.6	71.57	88.28	88.25	88.27
DCGAN	80.12	80.16	80.13	92.47	92.46	92.44
BERT-large	91.35	91.36	91.33	N/A		

Model accuracy of NN models trained with FP32, TF_AMP and Campo

Campo does not lose training accuracy!

FUTUREWEI Technologies

Conclusion

- The casting cost can outweigh the performance benefit of using lower precision!
- Campo uses a cost-aware graph rewriting strategy to achieve significant performance improvement in mixed-precision training.
- **Campo only casts to FP16 → There is potential for more performance enhancements with other lower-precision support in chips!**

CACHEW

Overview

Problem:

- Data preprocessing and model training/tuning cannot be scaled independently → tough to fix bottlenecks (model training runs on fast GPUs, input pipeline runs on slow CPUs)

Input processing Challenges

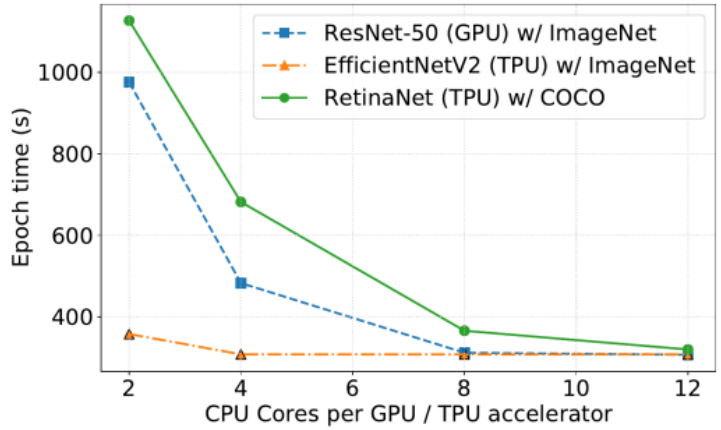
A. Waiting for batches costs time and money (expensive GPUs are waiting for data)

B. Small set of pipelines account for most computation

C. Preprocessing can consume more power than training

Opportunities

- A → **Scaling out**
- B / C → **Caching**



Current Landscape in ML Preprocessing

- Solutions for disaggregating input pipeline and model exist:
  - tf.data service from Google (open-sourced)
  - Data PreProcessing Service from Meta (closed-source)
- Fundamental mechanism is already there, **what is lacking is the functionality to perform these more automatically (resource allocation for data processing is complex)**

FUTUREWEI Technologies

Overview

Caching:

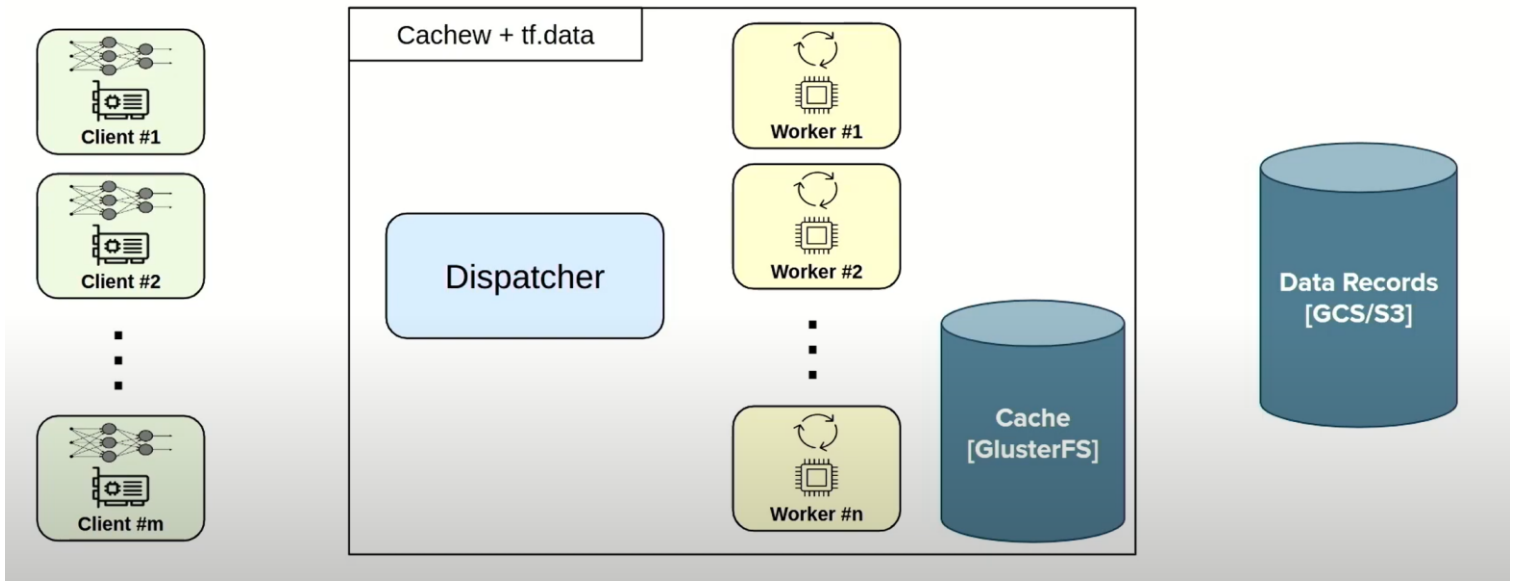
- Caching functionality already exist in many frameworks (it's not a research problem anymore)
- **Challenge: when and where does caching work in the input pipeline;** caching doesn't always make sense, i.e. reading from cache might be slower then recompute; even when it makes sense, it's not obvious where to utilize it in the input pipeline

As such, both scaling out and caching has to be done automatically → Contributions:

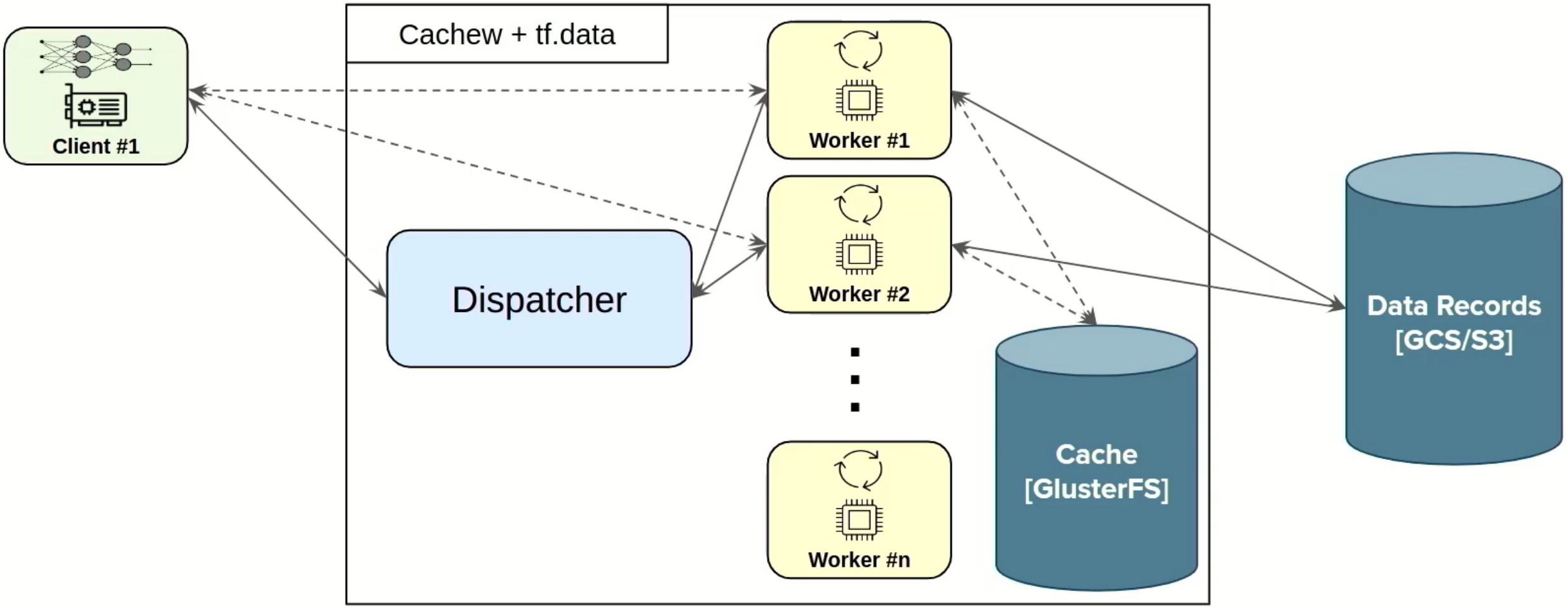
- **Autoscaling policy → How many resources** should be assigned to preprocessing?
- **Autocaching policy → When** and **Where** should data be cached?

System Architecture

- Builds on top of tf.data, because:
  - Disaggregation is already available
  - Open-source
  - Large-scale
  - Impactful
- GlusterFS for caching



System Architecture



Multi-tenancy **Disaggregation** **Autoscaling** **Autocaching**

FUTUREWEI Technologies

Code Annotation

Main features of Cachew:

- Multi-tenancy
- Disaggregation (inherited from tf.data)
- Autoscaling policy
- Autocaching policy: allows to decide if caching makes sense and if so where; also it can be used across jobs as well

Code Example

Operators added to the API

```
dataset = tf.data.TFRecordDataset(["file1", ...])
```
Read source data

```
dataset = dataset.map(parse).filter(filter_func)
          .autocache()
          .map(rand_augment)
          .autocache()
          .shuffle().batch()
          .autocache()
```
Define input pipeline logic

```
dataset = dataset.apply(distribute(dispatcherIP))
```
Process in service

```
for element in dataset:
  train_step(element)
```
Train model

Hints for the Cachew runtime → infer the throughput of the pipeline if you were to introduce caching at that location → ultimately chooses the option that is the fastest

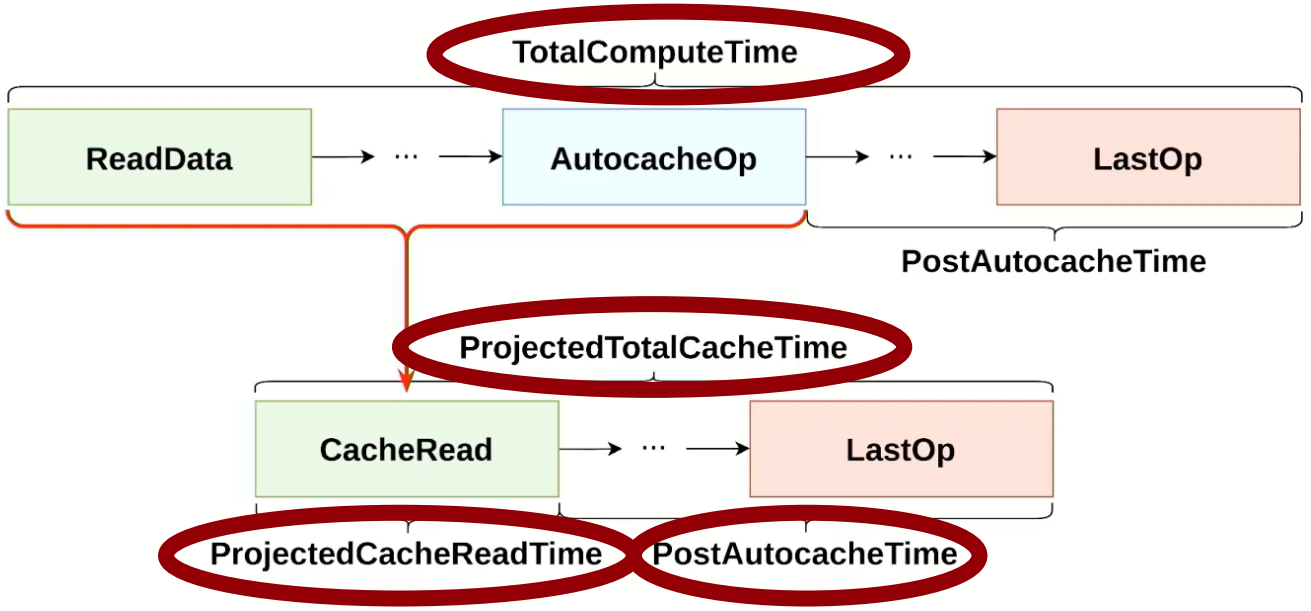
FUTUREWEI Technologies

Graph Representation

- **tf.graph's graph representation is inherited**
- **Input pipeline logic is internally represented as a graph**
- **Conveniently gathers relevant metrics in each node (used for auto-scaling and auto-caching mechanism)**
- **Can freely modify computation (done by workers) by changing the structure of the graph → Find the appropriate cache operation that enhances the throughput and replace AutoCacheOp with that operation (e.g. CacheGetOp)**

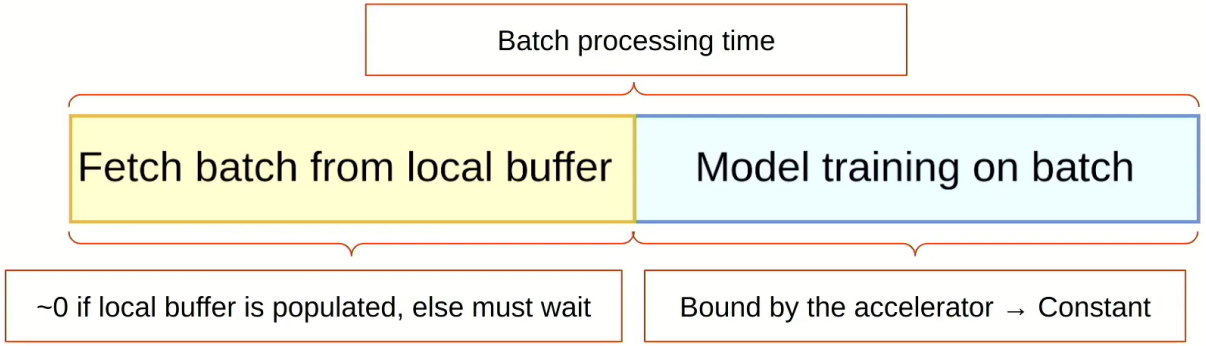
Autocaching Policy

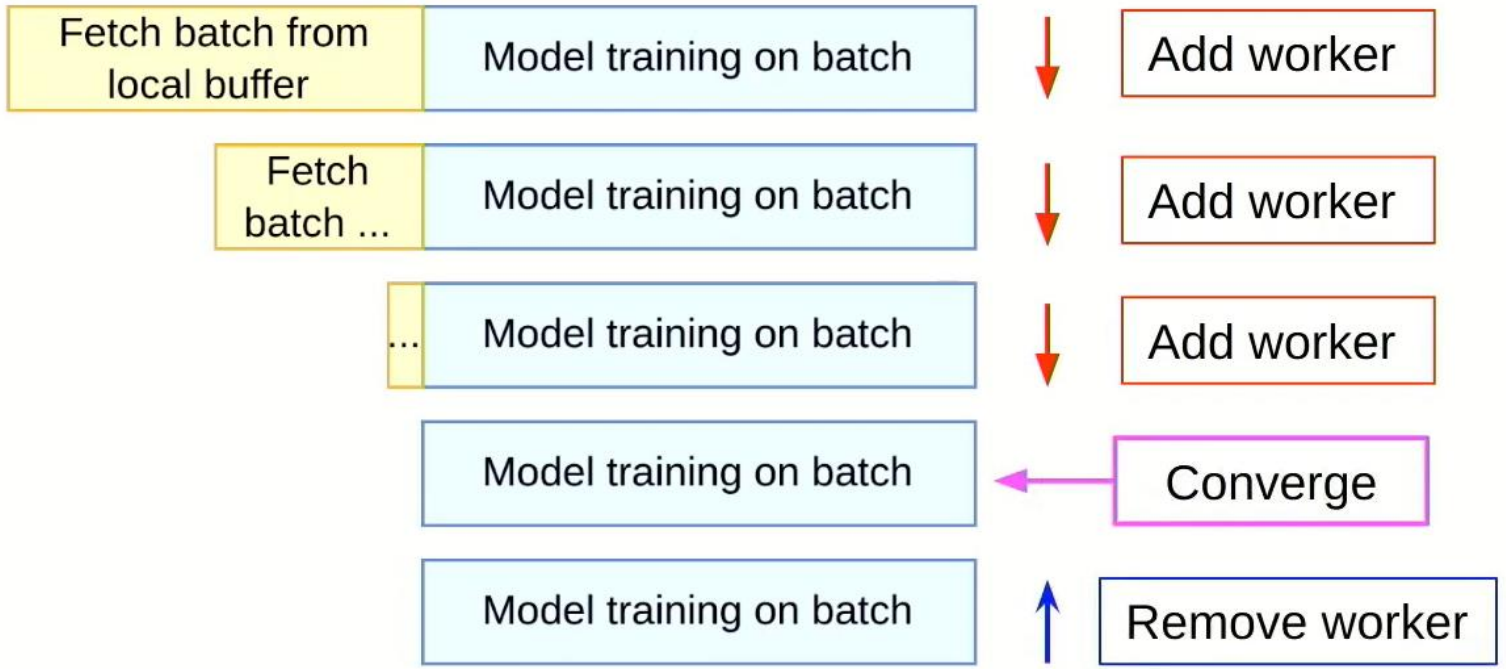
- Project throughput **for each autocache op** in input pipeline
- Choose option with highest throughput



Autoscaling Policy

- Two steps are executed when processing a batch
- Intuition: add workers to preprocessing until Batch Processing Time converges
- Add workers until convergence



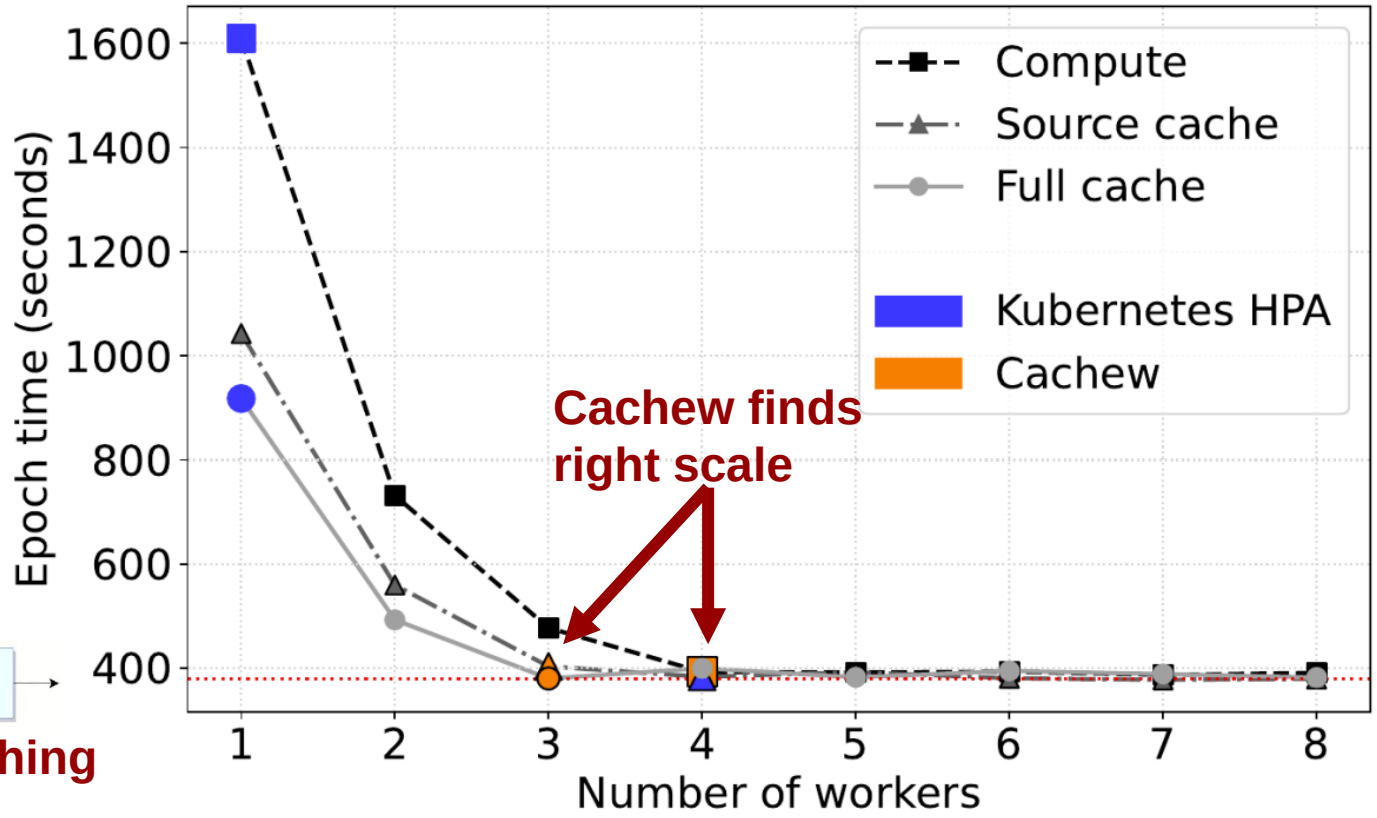


FUTUREWEI Technologies

Evaluation: Autoscaling Policy

Evaluation: Autoscaling Policy

- Deep Learning Image Classification workload:
  - ResNet input pipeline (GCE n2-standard-8 instance)
  - ResNet50 model (4 Nvidia V100 GPUs)
  - ImageNet dataset (approx. 140 GBs in GCS)



- **Input pipeline**

Source caching **Full caching**

- **Objective: Autoscale Policy decision and Kubernetes HPA decision**

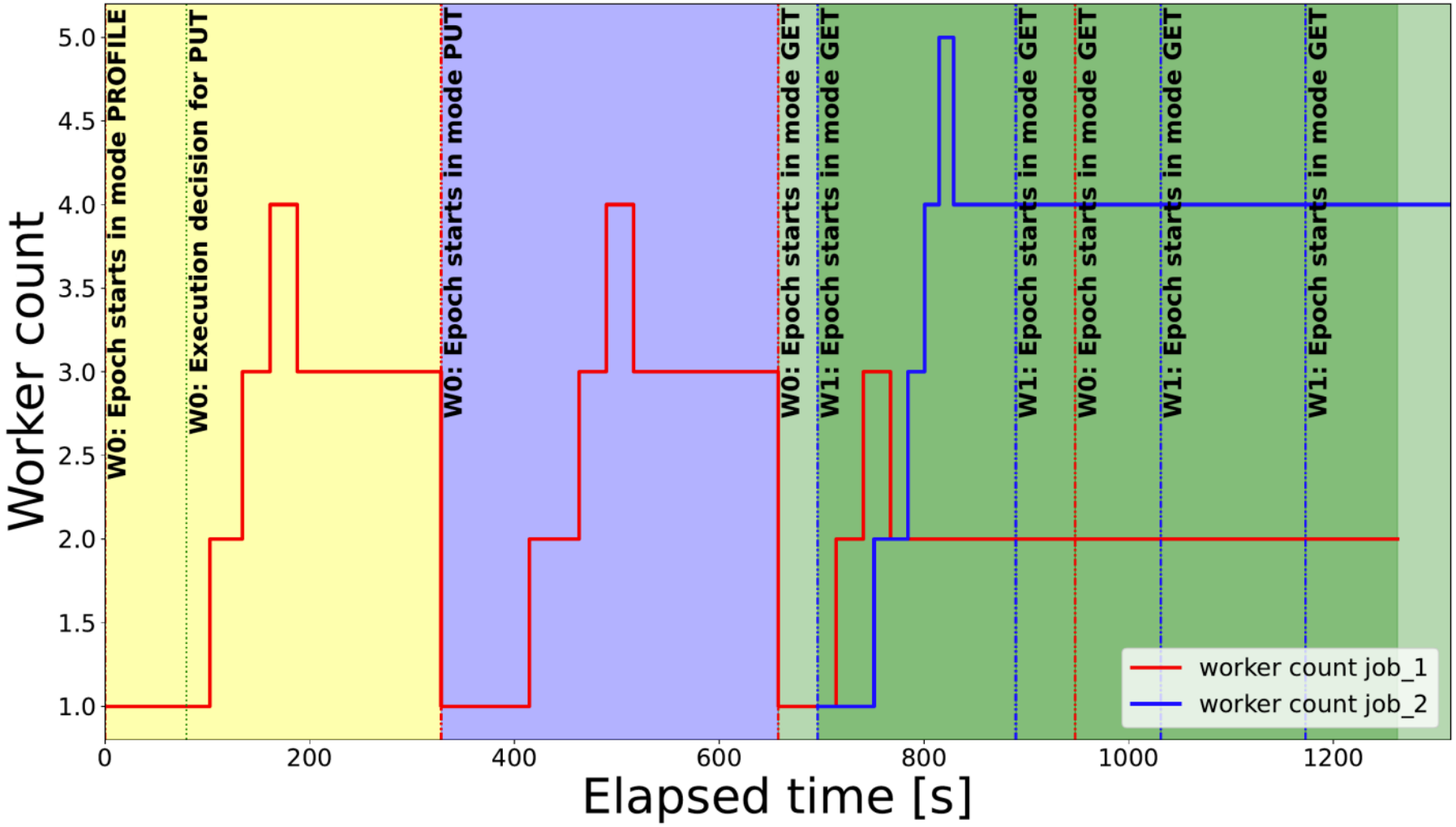
(* Source cache: *autocache* near the beginning of the input pipeline, immediately after a data read operator and before any data transformation operators)

(* Full cache: *autocache* at the end of the input pipeline, after all data transformations)

FUTUREWEI Technologies

Evaluation: Autoscaling & caching for Multiple Tenants

- Two jobs with 4 epochs each
- ResNet Input Pipeline with toy model
- Second job has twice the ingestion rate requirements
  - Requires twice the number of workers of job 1
- See how Cachew features work in multi-tenant environments



Conclusion

- Data preprocessing is essential in ML workloads
- Often bottleneck causing expensive accelerator stalls
- We propose Cachew, an Input-Pipeline-as-a-Service system:
  - Autocaching and Autoscaling Policies
  - Multi-tenancy support
- Open source: https://github.com/eth-easl/cachew
- Rich platform for future research

THANK YOU!