[Eurosys 23’]
SiloD: A Co-design of Caching and Scheduling for Deep Learning Clusters

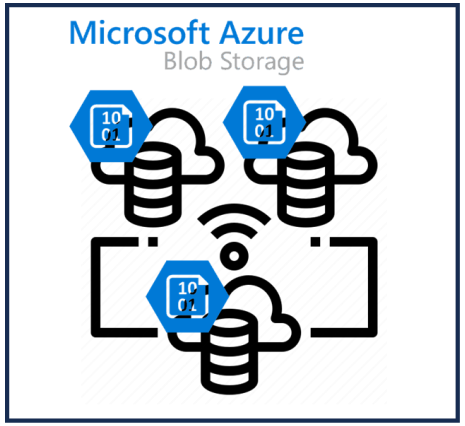
Minguk Choi

mgchoi@dankook.ac.kr

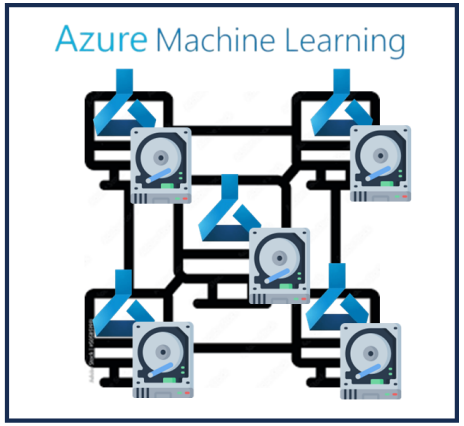
DANKOOK UNIVERSITY

Dankook University
System Software Laboratory

Background 1: Separation of Storage and GPU Clusters



Storage Cluster



Computing Cluster

- Deep learning training jobs read training data hosted in a separate cluster (storage services).
  - Decouples the storage service from the compute service
  - Modular and simple solution
- Bottleneck : Remote IO between compute and storage services
  - Leverage local storage of compute services as cache to alleviate the bottleneck.
- Cache & Compute services operates independently
  - Both do not aware each other.
  - Decoupled design leads to sub-optimal cluster performance.

Background 1: Separation of Storage and GPU Clusters

1. Increasingly large training datasets *[Tb. 1]*

2. Remote I/O as a bottleneck

(1) Performance enhancement *[Fig. 1]*
- In the last seven years, GPU: 125x, Remote I/O: 12x

(2) GPU ideal(aggregated) I/O demand
- 1923 MB/s for ResNet-50 with 8*A100 *[Tb. 2]*

- GPU(up to 200 Gbps) >> Remote I/O (up to 120 Gbps) *[Fig. 2]*

Dataset size	Year 2020	In 24 months
Task #1	25 TB	100 TB
Task #2	100 GB	1 TB
Task #3	100 GB	3 TB
Task #4	5 TB	10 TB
Task #5	1.5 TB	400 TB

Table 1. The size and growth of datasets for training at Microsoft.

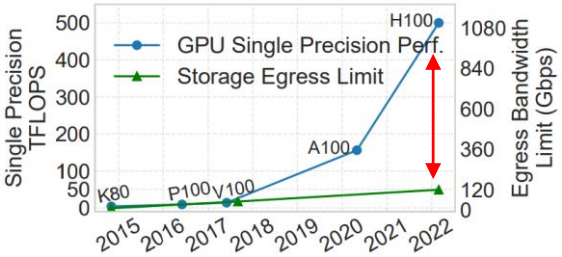


Figure 1. The trend of GPU perf. v.s. egress limits of cloud storage [10, 11].

GPU	Speed	IO
1*V100	1003 images/s	114 MB/s
1*A100	2930 images/s	333 MB/s
8*V100	7813 images/s	888 MB/s
8*A100	16925 images/s	1923 MB/s
1*Gaudi2	5325 images/s	614 MB/s

Table 2. Mixed-precision training and IO speeds of ResNet-50 on ImageNet.

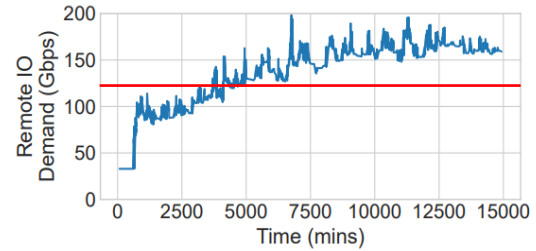


Figure 2. The IO demand of a 400-GPU (V100) cluster running a production trace. The peak IO achieves up to 200 Gbps.

Background 1: Separation of Storage and GPU Clusters

3. Cache subsystem for DL Training

(1) Built-in data loading library (CoorDL)
- isolated cache in training job's local storage with a static allocation
- cannot satisfy diverse demands on cache and remote I/O

(2) Distributed Cache (e.g., Alluxio, Quiver)
- shared cache in training cluster's local storage
- fast peer network as local storage (high-speed storage fabric)
[Fig. 3]

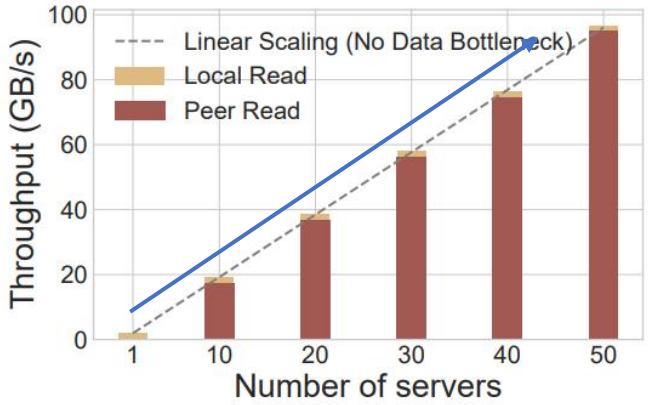


Figure 3. The throughput of distributed cluster running jobs with IO of 1923 MB/s (ResNet-50 on 8 A100s). All datasets are evenly distributed to all servers' cache. In n servers, each job will load $\frac{1}{n}$ data locally and $\frac{n-1}{n}$ data from peer servers.

Background 1: Separation of Storage and GPU Clusters

3. Cache subsystem for DL Training

(3) Built for general workloads
- do not exploit the characteristics of DL training:
  repetitive computation, predictable performance

- different scheduling objective:
  JCT, cluster throughput, fairness

- unawareness of the impact to other training jobs,
  missing global optimization [Fig. 4]

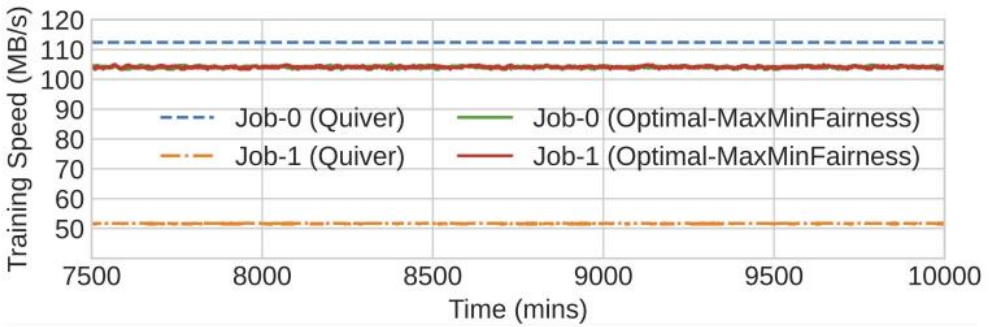
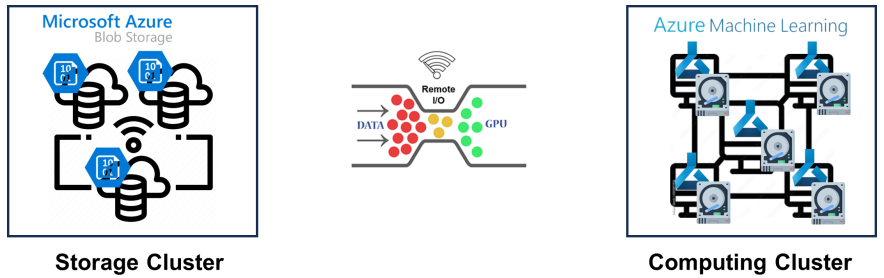


Figure 4. The training speeds of two ResNet-50 jobs training 1.36TB ImageNet-22k. The 2-GPU cluster has 1.4TB cache with 50 MB/s remote IO bandwidth. Quiver spends all cache to Job-0. The optimal max-min fair policy allocates half cache and remote IO to each of the jobs.

Summary of Background 1

Separation of Storage and GPU Clusters

- Bottleneck : Remote IO << GPU
  - Leverage local storage as cache
- Cache subsystem for DL Training
  - shared cache in training cluster's local storage
  - fast peer network as local storage
- Cache & Compute services operates independently
  - Both do not aware each other.
  - Decoupled design leads to sub-optimal cluster performance.



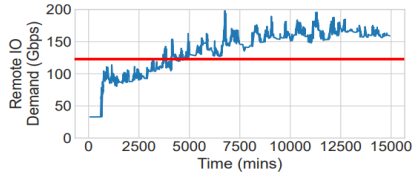


Figure 2. The IO demand of a 400-GPU (V100) cluster running a production trace. The peak IO achieves up to 200 Gbps.

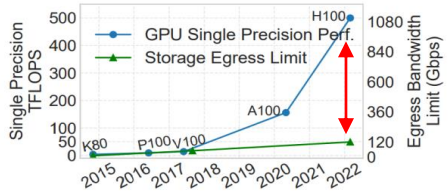
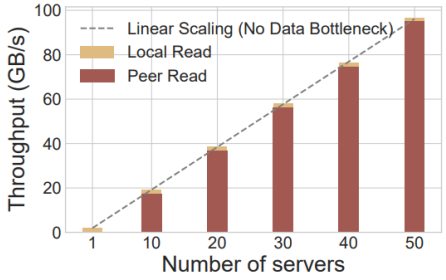
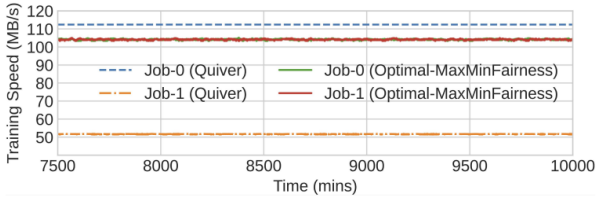


Figure 1. The trend of GPU perf. v.s. egress limits of cloud storage [10, 11].





Background 2: Opportunities of DL Training

1. Special data access pattern
within each epoch, access each data randomly, and exactly once.

2. Uniform Caching
(1) optimal for single training job

(2) all accessed data items are cached until the cache capacity
- constant and predictable cache hit ratio

(3) No eviction unless the cache capacity is reduced.
- cache eviction problem == cache space allocation problem

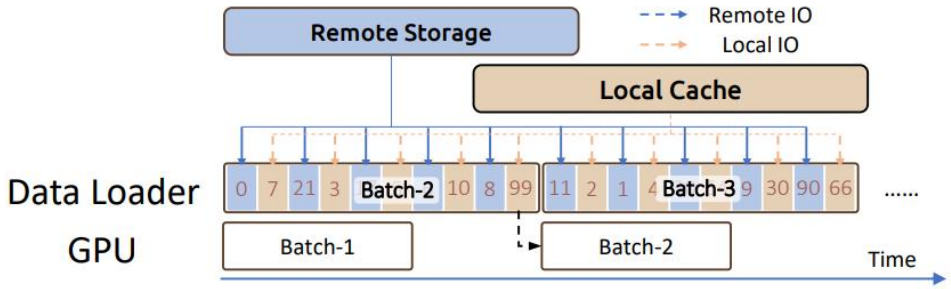


Figure 5. The pipeline of data loading and computation of deep learning training with uniform caching. Each data item has a unique ID. The missed data items are fetched from the remote storage. Because each epoch shuffles the data loading order, the expected cache hit ratio is uniform for all items. The example shows the training has a bottleneck on data loading.

Background 2: Opportunities of DL Training

3. Diverse cache and I/O demands

(1) In shared cluster, uniform caching may not be optimal.

(2) Cache efficiency = $\frac{f*}{d}$
$f^* = \mathrm{I/O}$ demand to achieve the ideal training speed
d = *the dataset size*

→ how much remote IO is saved if the entire dataset of a job is cached.

(3) Diverse cache efficiency
→ I/O demand ↑, dataset ↓ >> cache efficiency ↑

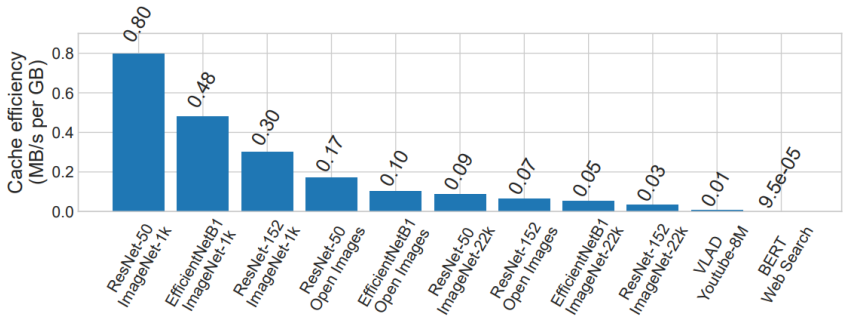


Figure 6. Cache efficiency on a V100 GPU. IO demand to achieve the ideal speed: ResNet-50 (114 MB/s), ResNet-152 (43 MB/s), EfficientNetB1 (69 MB/s), VLAD (10 MB/s), BERT (2 MB/s). The sizes of the datasets are listed in Table 1.

Dataset	Size	Model
ImageNet-22k [24]	1.36 TB	AlexNet [43], EfficientNetB0 [64], EfficientNetB1 [64], InceptionV3 [62], ResNet-50 [36], ResNet-152 [36]
Open Images [2]	660 GB	
ImageNet-1k [24]	143 GB	
Youtube-8M [13]	1.46 TB	VLAD [40]
Web Search	20.9 TB	BERT [25]

Table 4. Dataset and models used in the evaluation.

Background 2: Opportunities of DL Training

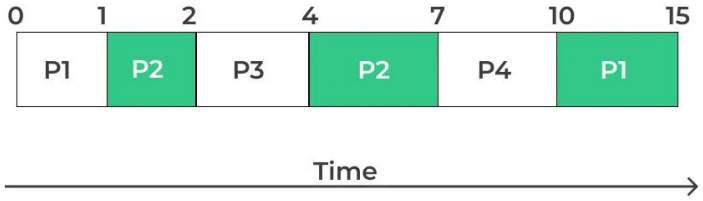
4. Co-designing cache and cluster scheduler

(1) Schedules as if the entire dataset of a job is cached.

(2) Cache and remote I/O could significantly affect performance when data loading is bottleneck.

(3) Necessary to co-design the cache and cluster scheduler to optimize scheduling objectives accurately

Shortest Job First (SJF) Preemptive Algorithm

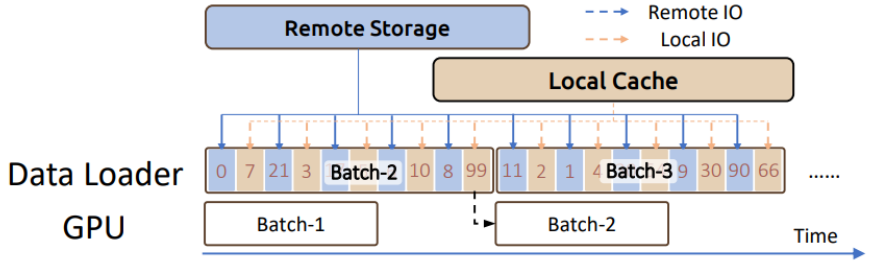


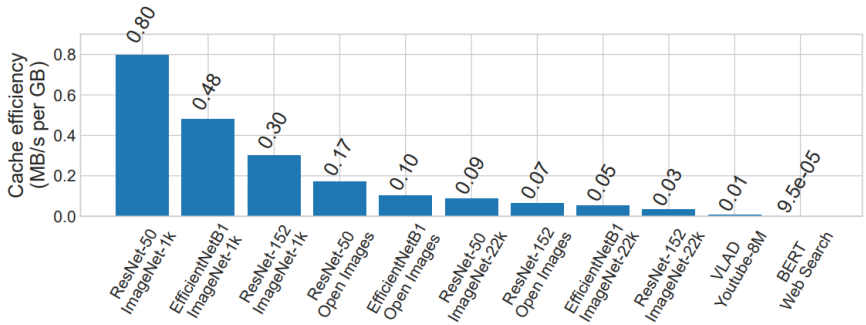
Process	Arrival Time	CPU Burst Time
P1	0	6
P2	1	
P3	2	2
P4	3	3

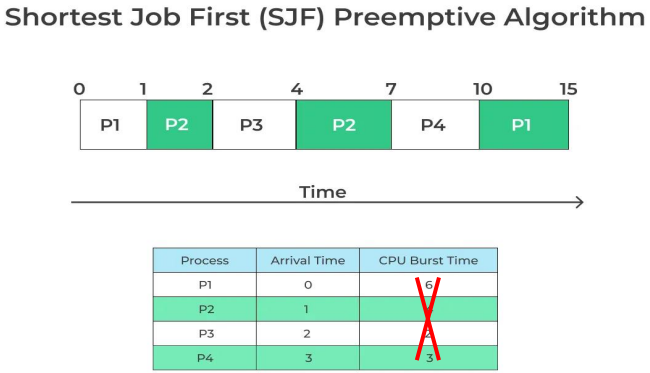
Summary of Background 2

Opportunities of DL Training

- Special data access pattern
  - within each epoch, access each data randomly, and exactly once.
- Uniform Caching
  - optimal for single training job, but not for shared cluster
- Diverse cache and I/O demands
  - Cache efficiency = $\frac{f*}{d}$
  - Diverse cache efficiency
- Co-designing cache and cluster scheduler
  - cluster schedules as if the entire dataset of a job is cached
  - cache and remote I/O significantly affect the training performance







Design: SiloD Overview

GOAL. Incorporate the diverse scheduling policies while exploiting the heterogeneous cache efficiency in a unified framework.

Design 1. Allocates compute and cache-related resources jointly to training jobs.

Design 2. Preserve original scheduling objectives.

```
def schedule(jobs, totalResource, perf):
  # perf(j, R): the performance estimator for
     estimating the compute throughput of job j under
     resource allocation R
  # totalResource: the total resource of the cluster

  SiloDPerf = lambda j, R: min(perf(j,R), IOPerf(j,R))
     # SiloD's enhanced performance estimator to
     jointly consider the impact of compute and storage
      resources

  alloc = Policy.Schedule(jobs, totalResource,
     SiloDPerf)
  return alloc
```

Algorithm 1. The workflow of SiloD. The underlined variables and functions are introduced/extended by SiloD and the others are inherited from existing schedulers.

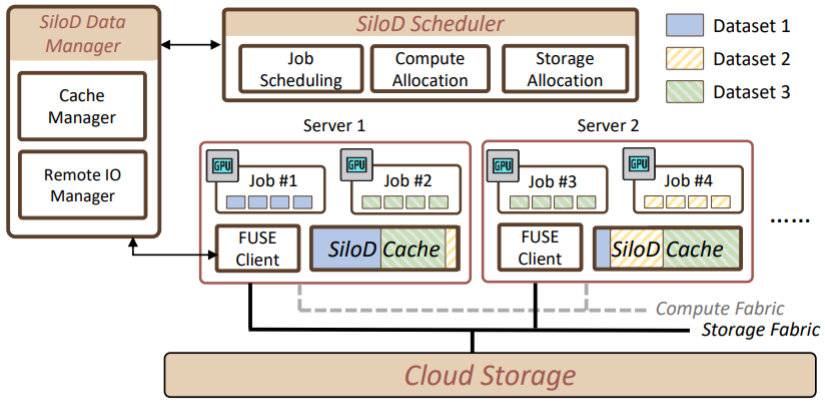
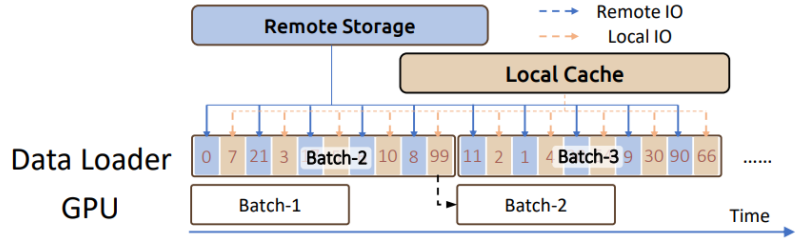


Figure 7. SiloD architecture.

Design 1: SiloD-Enhanced Performance Estimator



Design 1. Allocates compute and cache-related resources jointly to training jobs.

Consider uniform data access and pipelined execution of computation and data loading.

f : IOPerf (j, R) : jobs computation throughput
f^*: Perf (j,R) : jobs IO throughput
SiloDPerf = $\min\{f^*, f\}$: end-to-end throughput determined by bottle neck

c: allocated cache size
d: dataset size
b: remote IO demand of job

$\frac{c}{d}$: cache hit ratio

$1 - \frac{c}{d}$: cache miss ratio

$b = f \times \left(1 - \frac{c}{d}\right)$: job's remote IO demand

∵ data loading throughput × cache miss ratio

$f = \frac{b}{1 - c/d}$: a job's IO throughput

∵ remote IO bandwidth is limited, therefore throttling the remote IO to jobs when the sum of remote IO demand exceeds the bandwidth

SiloDPerf = $\min\{f^*, f\} = \min\left\{f^*, \frac{b}{1 - c/d}\right\}$

Cache Efficiency = $-\frac{\partial b}{\partial c} = \frac{f^*}{d}$

∵ the negative derivative of $b = f \times \left(1 - \frac{c}{d}\right)$

I/O demand ↑, dataset ↓ → cache efficiency ↑

Design 2: SiloD Policies

Design 2. Preserve original scheduling objectives.

(1) Shortest Job First (SJF): prioritizes the job with the least duration

score = weighted sum of resource demand of all resource types multiplied by its duration

The jobs with the least score will be scheduled first by the multi-resource SJF policy

$$\text{score} = \min_{\mathbf{R}} \sum_{t} w_t \cdot R_t \cdot \underbrace{\left(\frac{j.\text{numSteps} \cdot j.\text{stepDataSize}}{perf(j, \mathbf{R})} \right)}_{\text{job duration}},$$

$$\text{score} = \min_{\mathbf{R}} \sum_{t} w_t \cdot R_t \cdot \left(\frac{j.numSteps \cdot j.\text{stepDataSize}}{\texttt{SiloDPerf}(j, \mathbf{R})} \right).$$

SiloDPerf = $\min\{f^*, f\}$

$wt = \frac{1}{\text{totalResource[t]}}$: weight of the t-th resource type

R : a vector of allocation of all resource types

Rt : the allocation of the t-th resource type in R,

j.numSteps : the job j's total number of steps

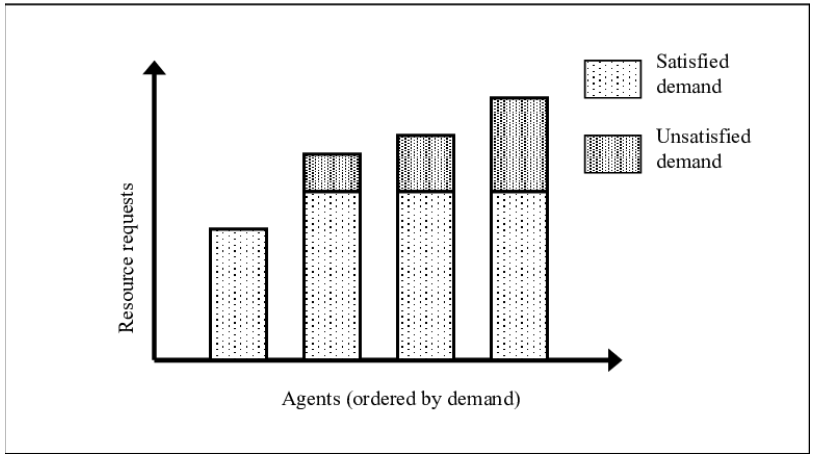
j.stepDataSize: the size of data consumed per step.

Design 2: SiloD Policies

Design 2. Preserve original scheduling objectives.

(2) Gravel: max-min fairness

maximizes the job with the least performance improvement over the equal resource division



$$\max_{R} \min_{j} \frac{perf(j, R[j])}{perf(j, R^{equal})}$$
$$s.t.\ Sum(R) \leq totalResource,$$

$$\max_{R} \min_{j} \frac{\texttt{SiloDPerf}(j, R[j])}{\texttt{SiloDPerf}(j, R^{equal})}$$
$$s.t.\ Sum(R) \leq totalResource.$$

$R[j]$: the resource allocated to job j and
$R\ equal$: the equal resource division among all jobs.

Design 2: SiloD Policies

Design 2. Preserve original scheduling objectives.

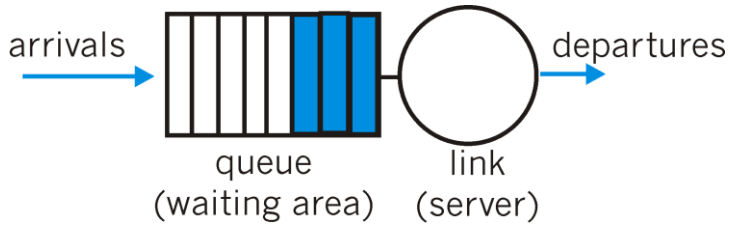
(3) A Greedy Policy for All Schedulers : do not rely on a performance estimator (e.g., FIFO)

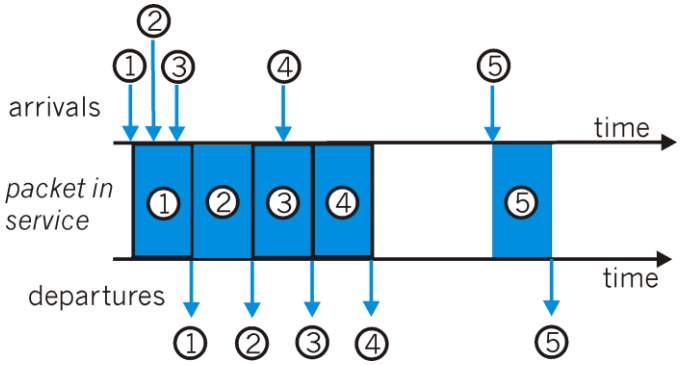
- directly leverages the cache efficiency

- allocating more cache to the most cache-efficient jobs

Algorithm 2 Greedy cache allocation policy

```
for job j in all jobs do
    j.CacheEfficiency = j.f* / j.datasetSize
for job j in descending order of j.CacheEfficiency do
    alloc.Cache[j] = min(j.datasetSize, totalCache)
    totalCache -= alloc.Cache[j]
return alloc
```





Summary of Design

GOAL. Incorporate the diverse scheduling policies while exploiting the heterogeneous cache efficiency in a unified framework.

Design 1. Allocates compute and cache-related resources jointly to training jobs.

SiloDPerf = $\min\{f^*, f\} = \min\left\{f^*, \frac{b}{1-c/d}\right\}$

→ end-to-end throughput determined by bottle neck
→ I/O demand ↑, dataset ↓ >> cache efficiency ↑

```
def schedule(jobs, totalResource, perf):
  # perf(j, R): the performance estimator for
     estimating the compute throughput of job j under
     resource allocation R
  # totalResource: the total resource of the cluster

  SiloDPerf = lambda j, R: min(perf(j,R), IOPerf(j,R))
     # SiloD's enhanced performance estimator to
     jointly consider the impact of compute and storage
      resources

  alloc = Policy.Schedule(jobs, totalResource,
     SiloDPerf)
  return alloc
```

Algorithm 1. The workflow of SiloD. The underlined variables and functions are introduced/extended by SiloD and the others are inherited from existing schedulers.

Design 2. Preserve original scheduling objectives.

(1) Shortest Job First (SJF): prioritizes the job with the least duration

$$\text{score} = \min_{\mathbf{R}} \sum_{t} w_t \cdot R_t \cdot \left(\frac{j.numSteps \cdot j.\text{stepDataSize}}{\texttt{SiloDPerf}(j, \mathbf{R})} \right).$$

(2) Gravel: max-min fairness

$$\max_{R} \min_{j} \frac{\texttt{SiloDPerf}(j, R[j])}{\texttt{SiloDPerf}(j, R^{equal})}$$
$$s.t.\ Sum(R) \leq totalResource.$$

(3) A Greedy Policy for All Schedulers : do not rely on a performance estimator (e.g., FIFO)

→ allocating more cache to the most cache-efficient jobs

Evaluation 1-1

Model	# of jobs	GPU/job	dataset size	# of epochs
ResNet-50	2	1 GPU	1.3TB	13
EfficientNetB1	2	1 GPU	1.3TB	10
BERT	1	4 GPU	20.9 TB	0.07

- Environment: 2 * 4-V100 VMs (=total 8 V100 GPUs)
- 2TB Storage Cache
- Remote IO bandwidth: 1.6 Gbps (200 MB/s)
- All dataset for each job is different
- # of epochs to let jobs run for 3,500 minutes

Evaluation 1-1

Model	# of jobs	GPU/job	dataset size	# of epochs
ResNet-50	2	1 GPU	1.3TB	13
EfficientNetB1	2	1 GPU	1.3TB	10
BERT	1	4 GPU	20.9 TB	0.07

- Environment: 2 * 4-V100 VMs (=total 8 V100 GPUs)
  - 2TB Storage Cache
  - Remote IO bandwidth: 1.6 Gbps (200 MB/s)
    - All dataset for each job is different
  - # of epochs to let jobs run for 3,500 minutes

- SiloD

Model	Cache Efficiency	Cache Usage	Remote IO
ResNet-50	87 MB/s/TB	1.3TB +0.7TB	0 + 52.6 MB/s
EfficientNetB1	53 MB/s/TB	-	2 ∗ 69 MB/s
BERT	0.4 MB/s/TB	-	8 MB/s

	Average JCT (relative error)		
	Real V100	Accelerated K80	Simulation
SiloD	**3366**	3339 (0.7%)	3403 (1.1%)
CoorDL	**4278**	4328 (1.1%)	4406 (3.0%)
Alluxio	**4378**	4519 (3.2%)	4484 (2.4%)
Quiver	**3609**	3534 (2.1%)	3592 (0.4%)

	Makespan (relative error)		
	Real V100	Accelerated K80	Simulation
SiloD	**3807**	3747 (1.5%)	3718 (2.3%)
CoorDL	**4870**	4925 (1.1%)	4918 (0.9%)
Alluxio	**5080**	5272 (3.7%)	4986 (1.8%)
Quiver	**3933**	3767 (4.4%)	3915 (0.4%)

Table 6. Average JCT and makespan (in minutes) in the 8-V100 experiment (bold), and relative error using the acceleration approach and the simulator.

Figure 9. The time-varying total job throughput in the 8-V100 experiment.

Evaluation 1-1

Model	# of jobs	GPU/job	dataset size	# of epochs
ResNet-50	2	1 GPU	1.3TB	13
EfficientNetB1	2	1 GPU	1.3TB	10
BERT	1	4 GPU	20.9 TB	0.07

- Environment: 2 * 4-V100 VMs (=total 8 V100 GPUs)
  - 2TB Storage Cache
- Remote IO bandwidth: 1.6 Gbps (200 MB/s)
  - All dataset for each job is different
- # of epochs to let jobs run for 3,500 minutes

Figure 9. The time-varying total job throughput in the 8-V100 experiment.

- CoorDL (data-loading library)
  - caches data for each job. unaware of the cache efficiency
  - wastes half of the total cache capacity (1TB) on BERT.
- Alluxio
  - LRU often evicts cached items that have not been read

Quiver's

caches one of the ResNet-50 jobs (1.3TB) and wastes the rest 0.7TB cache space.

Quiver's claim: jobs do not benefit if it cannot entirely fit into the cache.

Evaluation 2

- Environment: 96 V100s
- Remote IO bandwidth: 8 Gbps (1GB/s)
- Scheduling: FIFO
- Workload: a trace reported by Microsoft [41], (single-GPU + distributed multi-GPU training)
- More datasets and diverse combinations of models and datasets

Evaluation 2

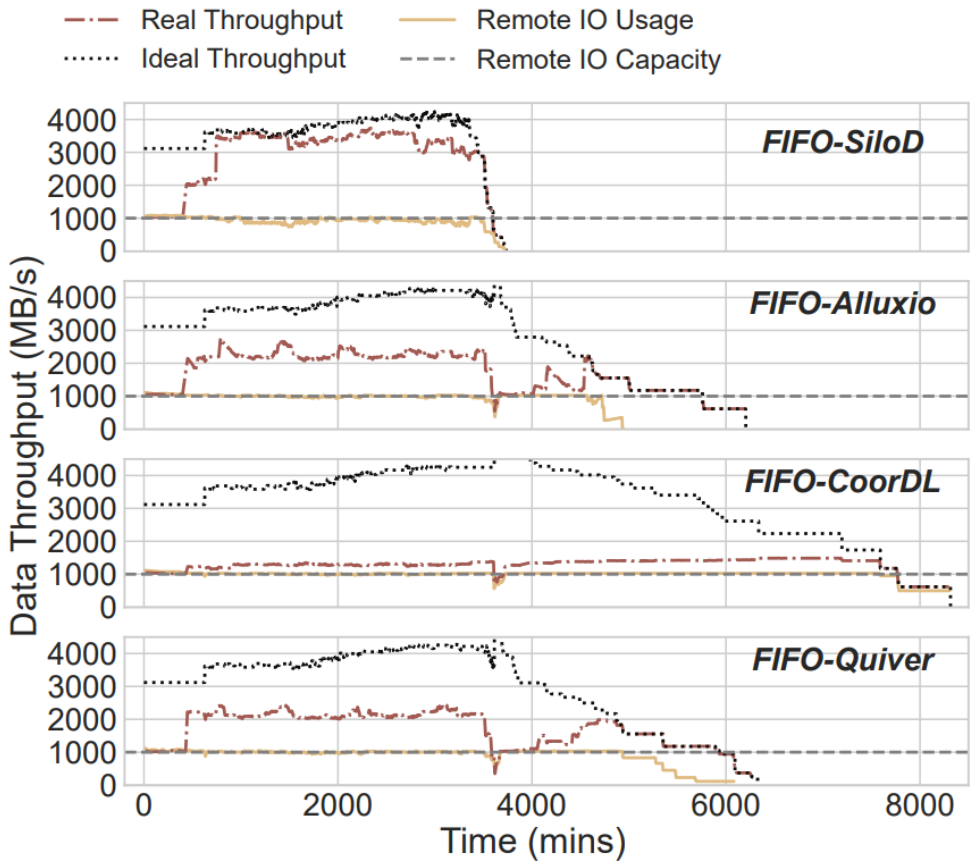
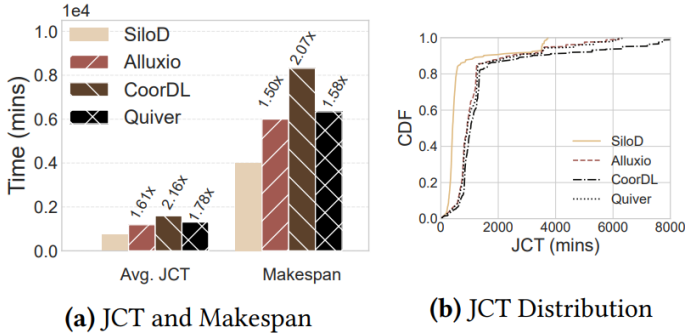


Figure 11. The total remote IO consumption, ideal training throughput, real training throughput in the 96-GPU cluster.



(a) JCT and Makespan **(b)** JCT Distribution

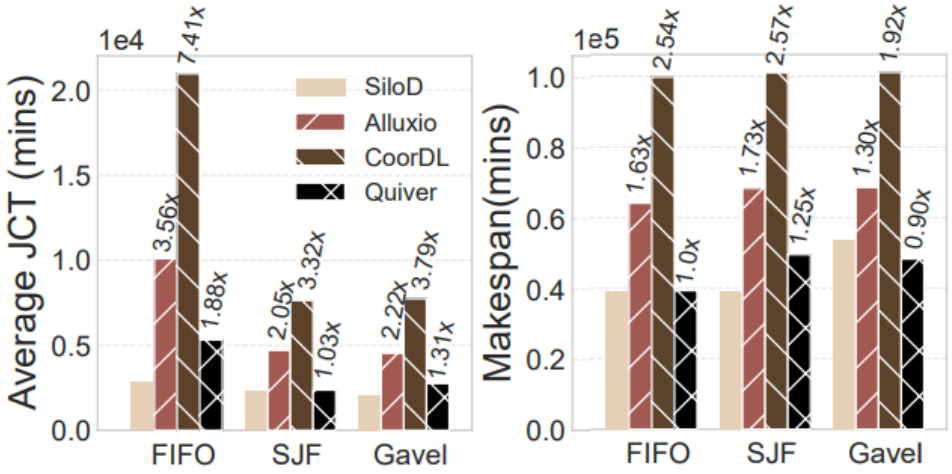
Figure 10. The average JCT, makespan and JCT distribution of the four policies in the FIFO-scheduled 96-GPU cluster.

- CoorDL (data-loading library)
  - benefits the least from cache.
  - only saves at most 490 MB/s remote IO
- Alluxio (LRU)
  - Fast jobs are more cache-efficient
  - Fast jobs evict the data of slow jobs that consume less IO.
- Quiver
  - Wasted cache space due to not supporting partial caching
  - Evict wrongly due to the unstable caching priority

Evaluation 3

- Environment: 400 V100s
- Remote IO bandwidth: 32 Gbps (1GB/s)
- Scheduling: FIFO
- Same workload with evaluation 2 with more jobs and longer running times (~ 4 weeks)

Evaluation 3



(a) Average JCT **(b)** Makespan

Figure 12. The performance of FIFO, SJF, Gavel using SiloD, Alluxio, CoorDL and Quiver in the 400-GPU simulation.

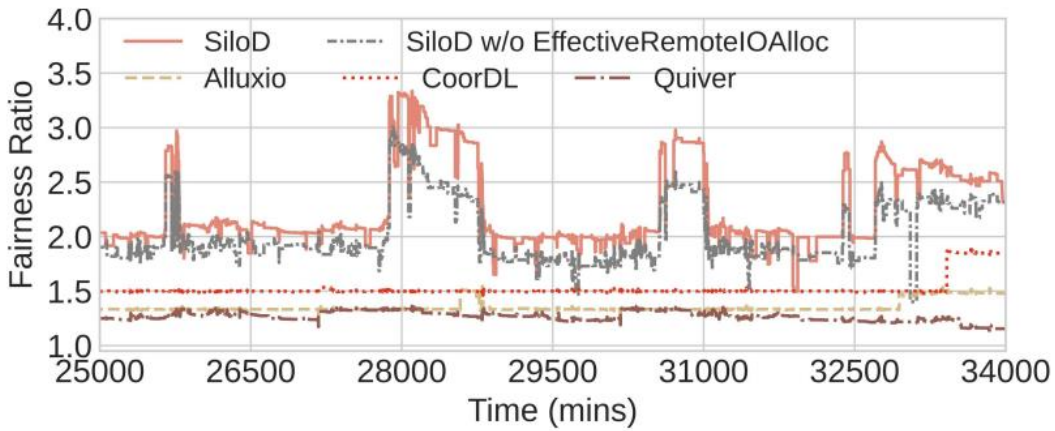


Figure 13. The fairness ratio over time in the 400-GPU clusters scheduled by Gavel. The higher the better.

Gavel on Quiver achieves slightly lower makespan than SiloD. Because Gavel optimizes for fairness, instead of makespan.

SiloD achieves the highest fairness ratio due to the co-scheduling directly optimizes the fairness objective

Conclusion

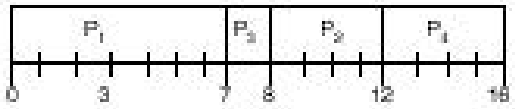
- SiloD is co-design of data caching and job scheduling based on the unique characteristics of deep learning training
- SiloD derive performance estimator to calculate the performance impact of both computation and storage.
- SiloD shows great improvement on their respective scheduling objectives.

Thank you

What is SJF

Example of Non-Preemptive SJF

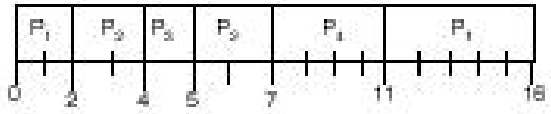
Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

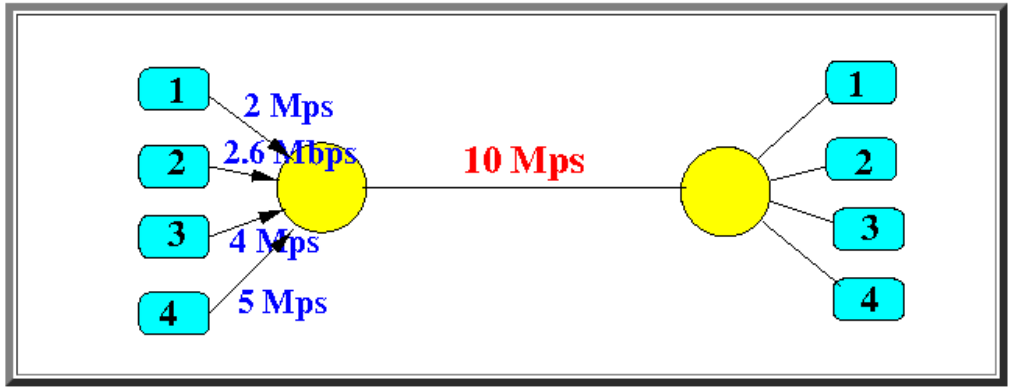


average waiting time = (9 + 1 + 0 +2)/4 = 3

What is min-max fairness?

Example of a Max-Min Fair Bandwidth assignment

- Consider the following 4 flows sharing a common bottle neck link:



Notes:

- The bottleneck link has a bandwidth of 10 Mbps
- There are 4 flows sharing the bottleneck link
- The demands of each flow is given in the figure

Ite Iteration 2:

- compute the fair share of each unsatisfied flow:

$$\frac{0.5 \text{ Mbps}}{3} = 0.16666 \text{ Mbps} \quad \text{(per flow)}$$

- Assignment:

1. Flow 2: 2.5 + 0.1 Mbps = 2.6 Mbps (because demand = 2.6 Mbps)
2. Flow 3: 2.5 + 0.16666 Mbps = 2.66666 Mbps
3. Flow 4: 2.5 + 0.16666 Mbps = 2.66666 Mbps

- Residual:
  - Unused bandwidth = 0.06666 Mpbs

Note:

Nc

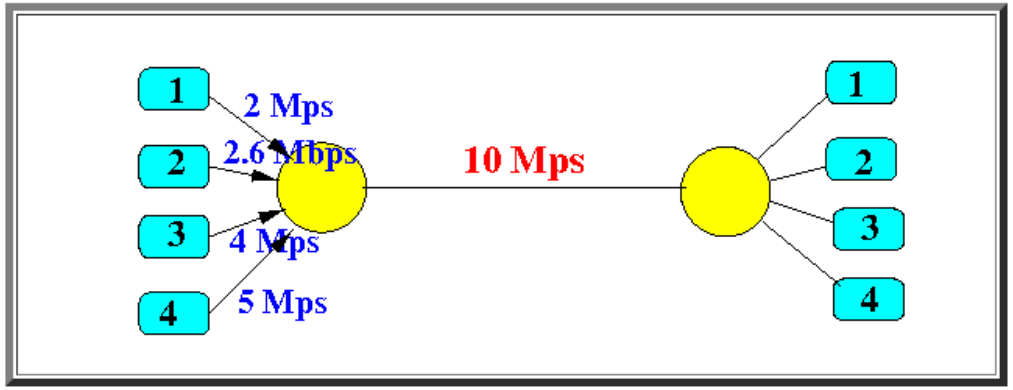
- After the *minimum* demand (i.e., flow 1 with 2 Mbps) has been maximized, the *second lowest* demand (i.e., flow 2 with 2.6 Mbps) is now maximized;

maximized

What is min-max fairness?

Example of a Max-Min Fair Bandwidth assignment

- Consider the following 4 flows sharing a common bottle neck link:



Notes:

- The bottleneck link has a bandwidth of 10 Mbps
- There are 4 flows sharing the bottleneck link
- The demands of each flow is given in the figure

Iteration 2:

- compute the fair share of each unsatisfied flow:

$$\frac{0.5 \text{ Mbps}}{3} = 0.16666 \text{ Mbps} \quad \text{(per flow)}$$

- Assignment:

1. Flow 2: 2.5 + 0.1 Mbps = 2.6 Mbps (because demand = 2.6 Mbps)
2. Flow 3: 2.5 + 0.16666 Mbps = 2.66666 Mbps
3. Flow 4: 2.5 + 0.16666 Mbps = 2.66666 Mbps

- Residual:
  - Unused bandwidth = 0.06666 Mpbs

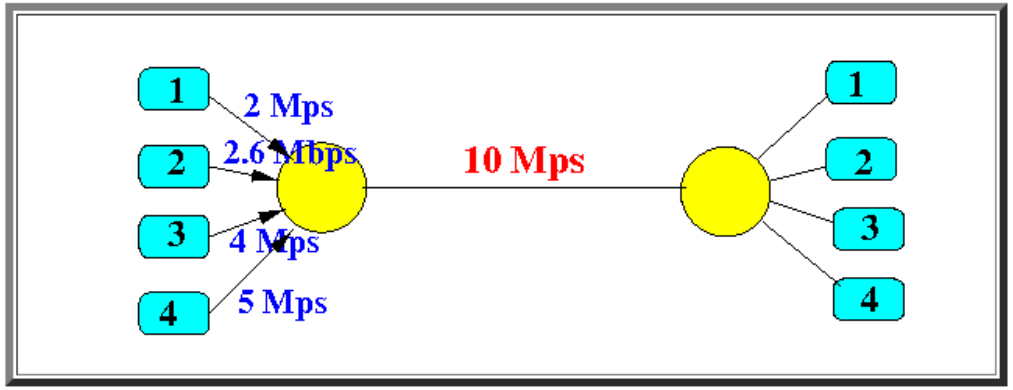
Note:

- After the *minimum* demand (i.e., flow 1 with 2 Mbps) has been maximized, the *second lowest* demand (i.e., flow 2 with 2.6 Mbps) is now maximized;

What is min-max fairness?

Example of a Max-Min Fair Bandwidth assignment

- Consider the following 4 flows sharing a common bottle neck link:



Notes:

- The bottleneck link has a bandwidth of 10 Mbps
- There are 4 flows sharing the bottleneck link
- The demands of each flow is given in the figure

Iteration 3:

- compute the fair share of each unsatisfied flow:

$$\frac{0.06666 \text{ Mbps}}{2} = 0.03333 \text{ Mbps} \quad \text{(per flow)}$$

- Assignment:

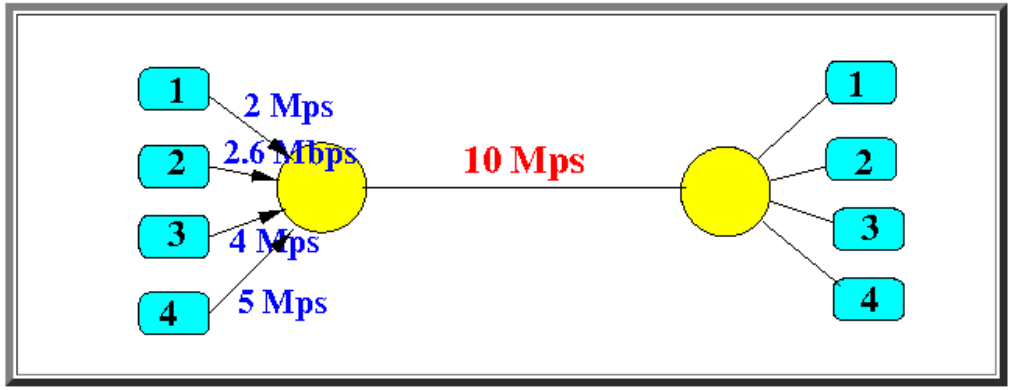
1. Flow 3: 2.66666 + 0.03333 Mbps = 2.7 Mbps
2. Flow 4: 2.66666 + 0.03333 Mbps = 2.7 Mbps

- Residual:
  - Unused bandwidth = 0.0 Mpbs

What is min-max fairness?

Example of a Max-Min Fair Bandwidth assignment

- **Consider** the following **4 flows** sharing a **common bottle neck link**:



Notes:

- The bottleneck link has a bandwidth of **10 Mbps**
- There are **4** flows sharing the bottleneck link
- The **demands** of **each flow** is given in the figure

- **Max-min fair assignment:**

1. Flow 1: **2 Mbps**
2. Flow 2: **2.6 Mbps**
3. Flow 3: **2.7 Mbps**
4. Flow 4: **2.7 Mbps**

Notice that:

- the ***lowest*** **demand** (= flow 1 with its **2 Mbps**) is **maximized**;
- the ***second lowest*** **demand** (= flow 2 with its **2.6 Mbps**) is **maximized**;
- the ***third lowest*** **demand** (= flow 3 with its **4 Mbps**) is **maximized**;

  (Note that **maximized** is ***not*** the same as **satisfied**. We gave **flow 3** the **highest possible assignment** that is **fair**)
- the ***fourth lowest*** **demand** (= flow 4 with its **5 Mbps**) is **maximized**;

  (Note that **maximized** is ***not*** the same as **satisfied**. We gave **flow 4** the **highest possible assignment** that is **fair**)