usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads

Deepak Narayanan and Keshav Santhanam, *Stanford University and Microsoft Research*; Fiodar Kazhamiaka, *Stanford University;* Amar Phanishayee, *Microsoft Research;* Matei Zaharia, *Stanford University*

https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads

Deepak Narayanan[†*], Keshav Santhanam[†*], Fiodar Kazhamiaka[†], Amar Phanishayee[⋆], Matei Zaharia[†]
[⋆]*Microsoft Research* [†]*Stanford University*

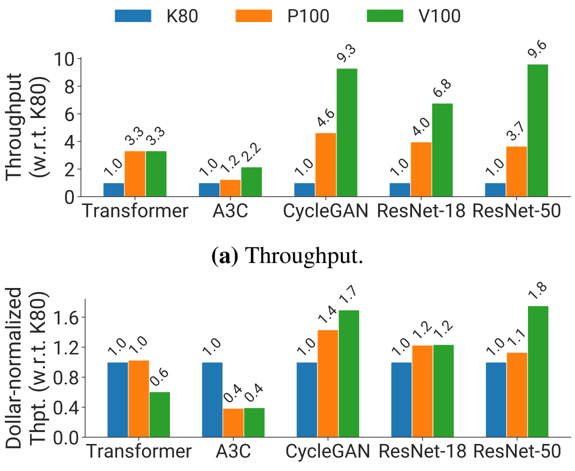
Abstract

Specialized accelerators such as GPUs, TPUs, FPGAs, and custom ASICs have been increasingly deployed to train deep learning models. These accelerators exhibit heterogeneous performance behavior across model architectures. Existing schedulers for clusters of accelerators, which are used to arbitrate these expensive training resources across many users, have shown how to optimize for various multi-job, multi-user objectives, like fairness and makespan. Unfortunately, existing schedulers largely do not consider performance heterogeneity. In this paper, we propose Gavel, a heterogeneity-aware scheduler that systematically generalizes a wide range of existing scheduling policies. Gavel expresses these policies as optimization problems and then systematically transforms these problems into heterogeneity-aware versions using an abstraction we call effective throughput. Gavel then uses a round-based scheduling mechanism to ensure jobs receive their ideal allocation given the target scheduling policy. Gavel's heterogeneity-aware policies allow a heterogeneous cluster to sustain higher input load, and improve end objectives such as makespan and average job completion time by 1.4× and 3.5× compared to heterogeneity-agnostic policies.

1 Introduction

As Moore's law comes to an end, specialized accelerators such as GPUs, TPUs, FPGAs, and other domain-specific architectures have emerged as an alternative to more general-purpose CPUs. These accelerators have been deployed to great effect [25, 35] to train state-of-the-art deep neural network (DNN) models for many domains, including language, image and video [14, 30, 31, 51, 55].

Consequently, users today must choose from a wide variety of accelerators to train their DNN models. For example, public cloud users can rent several generations of NVIDIA GPUs and Google TPUs from cloud providers [1–3]. Even organizations with private clusters have accumulated different accelerator types over time [34]; anecdotally, our research group has NVIDIA Titan V, Titan X, and P100 GPUs in its private cluster. Resources in these multi-tenant settings are typically arbitrated by a scheduler. GPU cluster schedulers such as Themis [40], Tiresias [28], AlloX [37], and Gandiva [58] thus need to decide how to allocate diverse resources to many users while implementing complex cluster-wide *scheduling policies*, optimizing objectives such as fairness or makespan. Unfortunately, choosing the most effective accelerator types in this context is difficult for three reasons:



(a) Throughput.

(b) Dollar-normalized.

Figure 1: Throughputs and dollar-normalized throughputs of training for various ML models. Dollar-normalized throughputs are computed by dividing the corresponding throughput by the relevant GCP on-demand price, The magnitude of speedup across GPU generations varies significantly across models.

Performance Heterogeneity. Commonly used models show heterogeneous performance behavior across accelerator types due to various architectural differences. For example, Figure 1a shows that a ResNet-50 model sees a nearly 10× speedup from an NVIDIA V100 GPU compared to a K80 GPU, while an A3C Deep Reinforcement Learning model only sees a 2× speedup. However, as shown in Figure 1b, the V100 is no longer the optimal choice for all models when we consider the number of samples trained per dollar – for many models, the older P100 GPU is competitive or cheaper on a per-dollar basis. Some scheduling policies can also benefit from splitting a job between *multiple* resource types: for example, minimizing a job's cost subject to a latency SLO (e.g., complete a job in 10 hours) might involve using a cheaper accelerator to begin training and then switching to a faster, more expensive device to meet the SLO. Thus, for even simple *single-job* settings, the choice of accelerator type is non-trivial and depends on *both* the job and the policy. This gets more complicated in *multi-job* settings as granting all jobs their

[*] Work done in part as interns at Microsoft Research.

preferred accelerator simultaneously might not be possible. Existing schedulers like Gandiva, Tiresias, and Themis do not consider this heterogeneous performance behavior.

Generality across Policies. Cluster operators might want to implement different scheduling policies based on their business goals, such as optimizing for time to complete a set of batch jobs (makespan), fairness for ad-hoc jobs, or more sophisticated *hierarchical* policies that divide resources among high-level entities (e.g., departments) using one policy, and then individual jobs within the entity using another [34]. In data analytics clusters, many job schedulers have support for hierarchical allocation policies [6, 7, 12, 59] already. The two recently proposed GPU schedulers that do consider heterogeneous resources, AlloX [37] and Gandiva$_{\text{fair}}$ [18], optimize for a single scheduling objective, and tightly couple their scheduling mechanism to that objective (e.g., max-min fairness). Thus, they cannot easily support the more sophisticated policies often used in practice.

Colocation and Placement Optimizations. To improve cluster utilization, existing GPU schedulers often deploy optimizations such as space sharing as in Gandiva [58], where multiple jobs can use the same accelerator concurrently, and placement sensitivity as in Themis and Tiresias [28, 40], which involves the careful placement of tasks in a distributed job to ensure good scaling performance. The performance benefits of these optimizations should be considered explicitly while optimizing for global scheduling objectives, since these optimizations are more effective when deployed in a heterogeneity-aware way. We show that explicit modeling for space sharing can improve objectives by 2.2× compared to Gandiva's ad-hoc approach.

In this paper, we present Gavel, a new cluster scheduler designed for DNN training in both on-premise and cloud deployments, that effectively incorporates heterogeneity in both hardware accelerators and workloads to generalize a wide range of existing scheduling policies. For example, Gavel can provide heterogeneity-aware versions of fair sharing / least attained service [28], FIFO, minimum makespan, minimum cost subject to SLOs, finish-time fairness [40], shortest job first, and hierarchical policies [12, 59].

Gavel's key observation is that many widely used scheduling policies, including hierarchical ones, can be expressed as *optimization problems* whose objective is a function of the jobs' achieved throughputs. For example, least attained service is equivalent to maximizing the minimum scaled throughput among the jobs, makespan is equivalent to minimizing the maximum duration (computed as the ratio of number of iterations to achieved throughput), and so on. Given the optimization problem for a scheduling policy, Gavel introduces a general way to transform the problem to make it heterogenity-, colocation- and placement-aware. In particular, Gavel changes the problem to search over a *heterogeneous allocation* for each job, the fraction of time spent in various resource configurations (e.g., 60% of time running alone on a V100 GPU and 40% of time space-sharing an A100 GPU with another job), and changes the throughput terms in the objective function to *effective throughput*, i.e. the average throughput of the job over the mix of resources in its allocation. Additional constraints need to be added to ensure that the returned allocation is valid. We show that Gavel's transformed optimization problems are efficient to execute even for clusters with hundreds of GPUs and jobs, and can support a wide range of policies. Many of these problems can be solved using a sequence of one or more linear programs.

Gavel's heterogeneity-aware allocations for each job need to be mapped to actual scheduling decisions (placement of jobs on specific resources in the cluster for a specified duration of time). To achieve this, Gavel uses a preemptive *round-based scheduling mechanism* to ensure that jobs receive resources in fractions similar to the computed target allocation. Gavel's scheduling mechanism needs to be able to schedule both distributed training jobs, which request multiple accelerators at once, as well as combinations of jobs running concurrently on a given accelerator due to space sharing.

Gavel makes these scheduling decisions transparently: it specifies an API between the scheduler and applications that allow jobs written in existing deep learning frameworks like PyTorch [48] and TensorFlow [13] to be moved between resources with minimal code changes, and uses a mechanism similar to Quasar [21] to estimate performance measurements of colocated jobs, which are needed as inputs to Gavel's policies, when not available *a priori*.

By explicitly considering performance heterogeneity, Gavel improves various policy objectives (e.g., average job completion time or makespan): on a smaller physical cluster, it improves average JCT by 1.5×, and on a larger simulated cluster, it increases the maximum input load a cluster can support, while improving objectives such as average job completion time by 3.5×, makespan by 2.5×, and cost by 1.4×.

To summarize, our main contributions are:

- A systematic method to convert existing cluster scheduling policies into equivalent policies that consider heterogeneity and colocation; these equivalent optimization problems are practical for current DNN clusters.
- A round-based scheduling mechanism to ensure that the cluster realizes the allocations returned by these policies.
- Generalizations of many existing policies in our framework that improve corresponding objectives.

Gavel is open sourced at `https://github.com/stanford-futuredata/gavel`.

2 Background

In this section, we provide a brief overview of DNN training (§2.1), and discuss performance optimizations used in existing schedulers that Gavel can help deploy more effectively (§2.2).

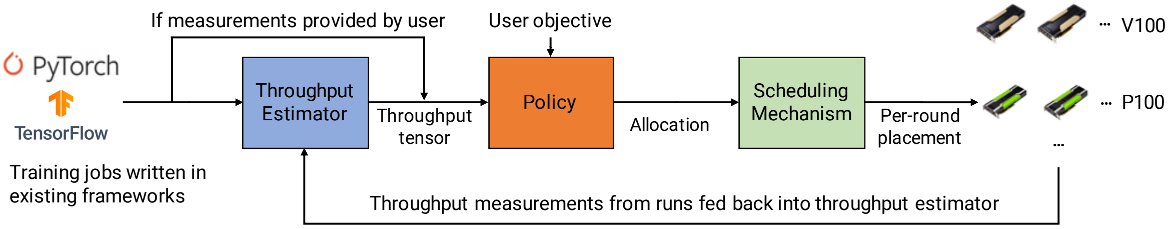

If measurements provided by user
User objective
PyTorch
TensorFlow
Training jobs written in existing frameworks
Throughput Estimator
Throughput tensor
Policy
Allocation
Scheduling Mechanism
Per-round placement
V100
P100
Throughput measurements from runs fed back into throughput estimator

Figure 2: Gavel overview. Jobs are written in frameworks like PyTorch or TensorFlow. Gavel's throughput estimator obtains performance measurements for each runnable job on each available accelerator type if necessary; its policy then computes an allocation that optimizes a user-specified objective such as fairness. Gavel's scheduling mechanism accepts this computed allocation as an input, and makes per-round placement decisions in proportions that faithfully mimic the computed allocation.

2.1 Deep Neural Network (DNN) Training

DNN training proceeds in iterations. In each iteration, the DNN processes a collection of inputs (called a minibatch) and subsequently updates the model parameters using gradients derived from the input minibatch. Each minibatch is typically of similar size, which means model training throughput using short profiling runs (order of minutes). Gavel leverages this fact in its throughput estimator. Jobs are typically fairly long-running (on the order of hours to days), and can be distributed over many workers [9, 58].

Modern DNN schedulers leverage the fact that DNN training is iterative to suspend and resume training at iteration boundaries [28, 58]; this ensures that jobs can be time multiplexed over the existing physical resources. The latest model parameters need to be checkpointed to stable storage when a job is suspended to ensure training progress is not lost. In this work, we show how *time sharing* should be deployed to optimize various single- and multi-job objectives.

2.2 Performance Optimizations

Prior work has shown that GPUs can be severely underutilized in multi-tenant clusters [34]; for example, average GPU utilization (measured as the percentage of GPU Streaming Multiprocessors active over time) was as low as 52% on a Microsoft cluster. Prior work has also shown the placement of tasks for a distributed training job can have significant impact on performance. Gavel can *optionally* deploy these optimizations systematically, as we show in §3.1.

Space Sharing. Smaller models often do not leverage the full computational capacity of modern GPUs. In such cases, concurrently executing multiple models on the same GPU using NVIDIA's Multi Process Service (MPS) or CUDA streams can help improve utilization [10, 47].

Placement Sensitivity. DNN models show heterogeneity in their distributed scaling behavior depending on the size of the tensors that need to be exchanged between workers during training: some models have compact weight representations and can scale well even when workers are not on the same server, while other models scale poorly when workers are spread over many servers. Existing schedulers like Tiresias use heuristics for placement sensitivity.

3 System Overview

Given a collection of jobs, Gavel arbitrates cluster resources (in the form of accelerators of different types) among the resident jobs, while optimizing for the desired cluster objective. This is accomplished in a two-step process: first, a *heterogeneity-aware policy* computes the fraction of time different jobs (and combinations) should run on different accelerator types to optimize the desired objective. These policies require as input the performance behavior (in terms of throughputs) for each job on each accelerator type, which can either be provided by the user, or can be measured on the fly by Gavel's throughput estimator. Allocations are intended to be respected only between allocation recomputation events; for example, if job 1 is much longer than job 2, the allocation will be recomputed once job 2 completes. Gavel can recompute its policy either when a *reset event* occurs (job arrives or completes, worker in the cluster fails), or at periodic intervals of time. Given the policy's output allocation, Gavel's *scheduling mechanism* grants jobs time on the different resources, and moves jobs between workers as necessary to ensure that the true fraction of time each job spends on different resources closely resembles the optimal allocation returned by the policy. Gavel's workflow is shown in Figure 2.

3.1 Heterogeneity-Aware Policies

Gavel expresses scheduling policies as optimization problems for various objectives of interest, such as fairness or makespan, and allocations as matrices that specify the fraction of wall-clock time a job should spend on each accelerator type *between* allocation recomputations. A matrix X can represent allocations on a single accelerator type (homogeneous setting), on multiple accelerator types (heterogeneous setting), as well as with other optimizations. Consider X^{example}:

$$\begin{array}{cc} & \begin{array}{ccc} V100 & P100 & K80 \end{array} \\ X^{\text{example}} = & \begin{pmatrix} 0.6 & 0.4 & 0.0 \\ 0.2 & 0.6 & 0.2 \\ 0.2 & 0.0 & 0.8 \end{pmatrix} \begin{array}{l} \text{job 0} \\ \text{job 1} \\ \text{job 2} \end{array} \end{array}$$

According to this allocation specified over three jobs and three accelerator types, job 0 should spend 60% of the time this

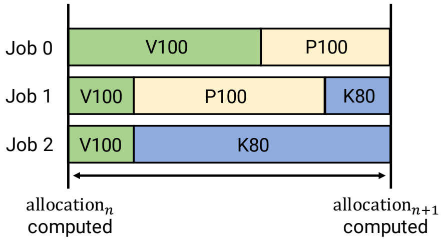
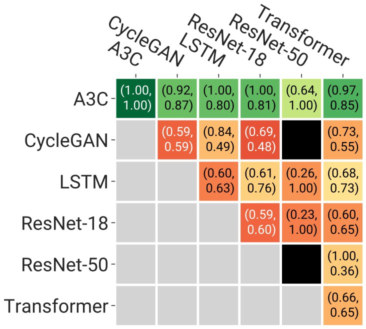


Figure 3: The *cumulative* time each job spends on accelerator types between allocation recomputations for allocation X^{example}.



	A3C	CycleGAN	LSTM	ResNet-18	ResNet-50	Transformer
A3C	(1.00, 1.00)	(0.92, 0.87)	(1.00, 0.80)	(1.00, 0.81)	(0.64, 1.00)	(0.97, 0.85)
CycleGAN		(0.59, 0.59)	(0.84, 0.49)	(0.69, 0.48)		(0.73, 0.55)
LSTM			(0.60, 0.63)	(0.61, 0.76)	(0.26, 1.00)	(0.68, 0.73)
ResNet-18				(0.59, 0.60)	(0.23, 1.00)	(0.60, 0.65)
ResNet-50						(1.00, 0.36)
Transformer						(0.66, 0.65)

Figure 4: Performance of several DNN models when run concurrently on a single P100 GPU. The cell at row i and column j reports the normalized throughput (iterations/second) achieved by co-located models i and j. Throughputs are normalized with respect to the throughput achieved by each model when run in isolation. Black squares show jobs that cannot co-locate due to memory constraints.

allocation is valid on a V100 GPU, and the remaining 40% of time on a P100 GPU. This is shown visually in Figure 3.

Gavel finds an optimal value for the matrix X given a policy expressed as an optimization problem. To construct the optimization problem for a given policy, Gavel requires a *throughput matrix* T with each job's throughput (in training iterations per second) on different accelerators. T_{mj} can be set to $-\infty$ if job m does not run on accelerator type j (for example, due to memory constraints).

Given T and X, we define the *effective throughput* of a model m as the time-weighted average throughput across accelerators and jobs. We denote this quantity $\text{throughput}_T(m,X)$ or simply $\text{throughput}(m,X)$ (dropping the T) for brevity. For allocations X without space sharing,

$$\text{throughput}(m,X) = \sum_{\substack{j\in \\ \text{accelerator types}}} T_{mj} \cdot X_{mj}$$

Different cluster scheduling policies can be expressed as optimization problems for X while maximizing or minimizing an appropriate objective function. Constraints need to be specified to ensure that X is a valid allocation. A hypothetical policy that maximizes total effective throughput looks like,

$$\text{Maximize}_X \sum_{m\in\text{jobs}} \text{throughput}(m,X)$$

Subject to the following constraints:

$$\begin{aligned} 0 \le X_{mj} &\le 1 & \forall(m,j) \quad (1) \\ \textstyle\sum_j X_{mj} &\le 1 & \forall m \quad (2) \\ \textstyle\sum_m X_{mj} \cdot \text{scale\_factor}_m &\le \text{num\_workers}_j & \forall j \quad (3) \end{aligned}$$

These constraints ensure that each job-worker allocation is non-negative and between 0 and 1 (equation 1), that the total allocation for a job does not exceed 1 (equation 2), and that the allocation does not oversubscribe workers (equation 3).

Space Sharing. Gavel's allocation matrices can also incorporate space sharing (SS). While previous work has used greedy algorithms for space sharing, we found that different pairs of DNN applications in practice have vastly different performance when colocated together, based on the resources they consume (Figure 4). When using space sharing, X needs to contain rows for each viable combination of jobs, and T needs to have throughputs of the job combinations, like:

$$T = \begin{array}{c} \\ \\ \\ \\ \end{array}\!\!\begin{array}{ccc} V100 & P100 & K80 \\ \end{array}$$

$$T = \begin{pmatrix} 40.0 & 20.0 & 10.0 \\ 15.0 & 10.0 & 5.0 \\ (20.0, 7.5) & 0.0 & 0.0 \end{pmatrix} \begin{array}{l} \text{job 0} \\ \text{job 1} \\ \text{jobs (0, 1)} \end{array}$$

The SS-aware allocation X dictates the fraction of time that each *job combination* should spend on each accelerator type.

We limit entries of T to combinations of at most 2 jobs; we found empirically that larger combinations rarely increase net throughput. Additionally, although the size of T grows quadratically with the number of jobs even with job combinations of size 2, we found that in practice we only need to consider combinations that actually perform well. We evaluate the scaling behavior of these SS-aware policies in §7.4.

Objectives in terms of $\text{throughput}(m,X)$ remain the same; however, $\text{throughput}(m,X)$ now needs to be computed to include the throughputs of co-located jobs:

$$\text{throughput}(m,X) = \sum_{\substack{j\in \\ \text{accelerator types}}} \sum_{k\in C_m} T_{kjm} \cdot X_{kjm}$$

The constraints need to be slighly modified as well to ensure that X is a valid allocation in this new regime:

$$\begin{aligned} 0 \le X_{kj} &\le 1 & \forall(k,j) \\ \textstyle\sum_{k\in C_m} \sum_j X_{kj} &\le 1 & \forall m \\ \textstyle\sum_k X_{kj} \cdot \text{scale\_factor}_m &\le \text{num\_workers}_j & \forall j \end{aligned}$$

C_m is the set of all job combinations that contain job m.

Placement Sensitivity. Similarly, Gavel's allocation matrices can also be extended to incorporate placement sensitivity. The observed throughput for distributed jobs depends on the location of tasks, as well as the model and accelerator type (slower workers are less likely to be communication-bound,

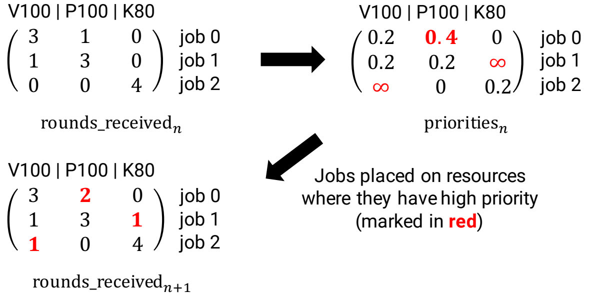


Figure 5: Priorities are used to move the received allocation towards the intended allocation (in this case, X^{example}). priorities$_n$ is computed as $X/\text{rounds\_received}_n$ (element-wise division).

which means consolidation of tasks is less effective). We can make our policies *placement-sensitive* by considering the performance of distributed jobs in: 1) a consolidated setting, where as many accelerators are on the same server as possible (for example, 8 GPUs per server if using 8-GPU servers), and 2) an unconsolidated setting, where accelerators are on independent servers. These are extreme points in the placement space, and are upper and lower bounds on performance. We can model this in our policies by having two different worker types (consolidated and unconsolidated) with corresponding throughput values in T and allocation values in X.

3.2 Round-based Scheduling Mechanism

After computing the optimal allocation, Gavel's next step is to assign jobs (or job combinations, in the case of SS) to accelerator types while matching the optimal allocation as closely as possible. That is, to realize the allocation X^{example} above, the scheduling mechanism needs to make sure that in the time period where jobs 0, 1, and 2 are the only three runnable jobs in the cluster, jobs should receive resources according to their computed optimal time fractions.

To do this, the scheduler computes a priority score for every job and accelerator type combination that is high when a job has received a smaller time fraction than the optimal allocation. Scheduling is performed in rounds; in each round, the scheduler runs jobs in decreasing priority order, while ensuring that a given job is not scheduled on multiple workers (or accelerators) in a given round. This is shown in Figure 5. Priorities are updated as rounds complete. We have found empirically that round durations of around 6 minutes allow Gavel to effectively approximate the ideal allocation (§7.5).

3.3 Throughput Estimator

To estimate the throughputs of concurrent jobs (e.g., in the case of space sharing), Gavel employs a throughput estimator, similar to those found in prior work such as Quasar [21]. Gavel's throughput estimator maps a new job to a set of pre-profiled reference jobs. The throughputs of the closest reference job can then be used as the initial performance estimate for the new job's combinations. For individual jobs, the throughput estimator is not needed, since throughputs can be estimated on the fly as jobs run on different resource types.

3.4 Limitations and Non-Goals

While Gavel exposes a flexible API that supports a variety of policies and objectives, we do not propose new scheduling policies or performance optimizations in this work. Instead, Gavel's main goal is to determine how best to share resources amongst many different users and jobs in a heterogeneity-aware way, while supporting many existing cluster-wide objectives. Gavel accomplishes these goals with a policy framework that easily allows policies to be made heterogeneity-, colocation-, and placement-aware (§4), a reusable scheduling mechanism (§5), and a narrow scheduler API that allows users to deploy their applications with minimal code changes (§6).

4 Scheduling Policies

In this section, we show how various scheduling policies such as max-min fairness (Least Attained Service or LAS) and multi-level fairness can be expressed as optimization problems in terms of effective throughput. We describe some properties of the resulting heterogeneity-aware allocations at the end of this section.

4.1 Max-Min Fairness as an Optimization Problem

The classical Least Attained Service (LAS) policy, used by Tiresias [28], implements max-min fairness across active users in the cluster, by round-robining resources across jobs according to the total number of accelerator hours consumed. This can be modified into a weighted max-min fairness policy with per-user weights w_m. On a homogeneous cluster, if a job m with weight w_m receives a fraction X_m (which is a scalar since there is only one resource type), LAS can be expressed as the following optimization problem:

$$\text{Maximize}_X \min_m \frac{1}{w_m} X_m$$

We need to add an additional constraint to ensure that the cluster is not overprovisioned ($\sum_m X_m \leq 1$).

However, this vanilla LAS policy is not fair in a heterogeneous setting; jobs might see unequal reductions in throughput due to variations in performance across accelerator types. For example, giving one job a K80 and another job a V100 would equalize their number of resources, but could result in very low performance for the job with the K80.

To compute a more fair allocation, we can compute max-min fairness over the weighted normalized effective throughputs, as defined in §3.1. Let X_m^{equal} be the allocation given to job m assuming it receives equal time share on each worker in the cluster. For example, if the cluster had 1 V100 and 1 K80, $X_m^{\text{equal}} = [0.5, 0.5]$. X_m^{equal} scales the effective throughputs to make them comparable across jobs.

$$\text{Maximize}_X \min_m \frac{1}{w_m} \frac{\text{throughput}(m, X)}{\text{throughput}(m, X_m^{\text{equal}})}$$

Policy	Description
Makespan	Minimize time taken by batch of jobs.
LAS [28]	Max-min fairness by total compute time.
LAS w/ weights	Max-min fairness with weights.
Finish Time Fairness [40]	Maximize minimum job speedup.
FIFO	First in, first out.
Shortest Job First	Minimize time taken by shortest job.
Minimize cost	Minimize total cost in public cloud.
Minimize cost w/ SLOs	Minimize total cost subject to SLOs.
Hierarchical [59]	Multi-level policy: FIFO, fairness, etc.

Table 1: Policies that can be expressed in Gavel.

As specified in §3.1, additional constraints need to be specified to ensure that allocations are valid.

As an example, consider 3 jobs which benefit differently when moved from a K80 GPU to a V100 GPU:

$$T = \begin{array}{ccc} V100 & K80 & \\ 40.0 & 10.0 & \text{job 0} \\ 12.0 & 4.0 & \text{job 1} \\ 100.0 & 50.0 & \text{job 2} \end{array}$$

Solving the above optimization problem with $w_m = 1$, and a cluster with 1 V100 and 1 K80 yields the following allocation:

$$X^{\text{het.}} = \begin{array}{ccc} V100 & K80 & \\ 0.45 & 0.0 & \text{job 0} \\ 0.45 & 0.09 & \text{job 1} \\ 0.09 & 0.91 & \text{job 2} \end{array}$$

Jobs receive about 10% higher throughput compared to an allocation where every user is given $1/n$ of the time on each accelerator (here, $n = 3$), also called an *isolated allocation* [26].

Fairness policy objective functions need to be modified to take into account muti-resource jobs with $\text{scale\_factor}_m > 1$, since these multi-resource jobs occupy a larger share of the cluster per unit time. An easy way to do this is to multiply the max-min objectives from before by scale_factor_m. Concretely, the LAS objective from before now becomes,

$$\text{Maximize}_X \min_m \frac{1}{w_m} \frac{\text{throughput}(m, X)}{\text{throughput}(m, X_m^{\text{equal}})} \cdot \text{scale\_factor}_m$$

4.2 Other Policies as Optimization Problems

We can express many other common cluster scheduling policies, some proposed by recent papers, using $\text{throughput}(m, X)$; we list these policies in Table 1. Most of these policies can be expressed using a single linear program, with a few exceptions: the cost policies are formulated as a linear-fractional program [8], which can be reduced to a sequence of linear programs. These optimization problems yield corresponding heterogeneity-aware allocations. The optimal allocation can be computed using off-the-shelf solvers.

Minimize Makespan. The makespan minimization policy tries to complete all active jobs as soon as possible. Gandiva uses a version of this policy to finish higher-level tasks such as hyperparameter tuning and AutoML, which involve training a large number of variants of a model. If num_steps_m is the number of iterations remaining to train model m, then the makespan is the maximum of the durations of all active jobs, where the duration of job m is the ratio of the number of iterations to $\text{throughput}(m, X)$ (expressed in iterations / second). Overall, this can be framed as,

$$\text{Minimize}_X \max_m \frac{\text{num\_steps}_m}{\text{throughput}(m, X)}$$

Minimize Finish-Time Fairness (Themis). Themis [40] proposes a new metric called finish-time fairness (represented as ρ), which is the ratio of the time taken to finish a job using a given allocation and the time taken to finish the job using $1/n$ of the cluster (X^{isolated}), assuming n users using the cluster. This can be expressed in terms of $\text{throughput}(m, X)$ as follows (num_steps_m is the number of iterations remaining to train model m, t_m is the time elapsed since the start of training for model m, and t_m^{isolated} is the hypothetical time elapsed since the start of training if model m had $1/n$ of the cluster to itself),

$$\rho_T(m, X) = \frac{t_m + \frac{\text{num\_steps}_m}{\text{throughput}(m, X)}}{t_m^{\text{isolated}} + \frac{\text{num\_steps}_m}{\text{throughput}(m, X^{\text{isolated}})}}$$

The final optimization problem is then,

$$\text{Minimize}_X \max_m \rho_T(m, X)$$

FIFO. The First-In-First-Out (FIFO) policy schedules jobs in the order they arrive. In a heterogeneous regime, jobs should be placed on the fastest available accelerator type. Mathematically, we can write this as maximizing the throughput of job m relative to its throughput on the fastest type ($\text{throughput}(m, X^{\text{fastest}})$). Assuming that jobs are enumerated in order of their arrival time (m arrived before $m+1$), a FIFO allocation can be computed with the following objective:

$$\text{Maximize}_X \sum_m \frac{\text{throughput}(m, X)}{\text{throughput}(m, X^{\text{fastest}})} (M - m)$$

where M is the total number of jobs.

Shortest Job First. The Shortest Job First policy finds the allocation that minimizes the duration of the shortest job,

$$\text{Minimize}_X \min_m \frac{\text{num\_steps}_m}{\text{throughput}(m, X)}$$

Minimizing Total Cost and Cost subject to SLOs. We can express policies for deployments that use elastic public cloud resources. Since cloud VMs are charged on a per-time basis, we can express policies that explicitly optimize for total cost, speed, or both.

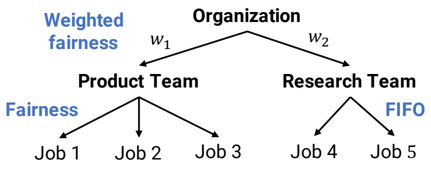


Figure 6: Example of a hierarchical policy: weighted fairness across two entities: a product and research team, fairness across jobs within the product team, and FIFO within the research team.

Consider a simple policy that maximizes total throughput,

$$\text{Minimize}_X \sum_m \text{throughput}(m, X)$$

The above policy can be extended to incorporate cost by optimizing the following cost-adjusted objective,

$$\text{Maximize}_X \frac{\sum_m \text{throughput}(m, X)}{\sum_m(\sum_j \text{cost}_j \cdot X_{mj})}$$

where cost_j is the cost of accelerator type j. The numerator in the above objective is the time-averaged effective throughput, and the denominator is the time-averaged cost. When using space sharing, care must be taken to not double count the cost of instances running job combinations (all jobs in a job combination derive value in the form of some throughput).

Jobs can have time SLOs as well, e.g., certain high-priority jobs might need to complete every 12 hours. We can add additional constraints: given SLO_m for each model m (models without SLOs can have $\text{SLO}_m = \infty$),

$$\text{throughput}(m, X) \geq \text{num\_steps}_m / \text{SLO}_m$$

4.3 Hierarchical Scheduling Policies

Modern cluster schedulers do not only deploy "single-level" policies. Hierarchical policies are common [6, 12, 59]: a large organization might share a single physical cluster among many sub-organizations (or entities) using a fairness policy. In turn, each entity can share resources among individual jobs according to a distinct per-entity policy, such as per-user fairness or FIFO. We give an example in Figure 6, where a research and product team share the same physical cluster. The research team runs ad-hoc experiments that can be executed in FIFO order, but the product team needs to ensure that all its jobs receive a fair share of the cluster.

Gavel can currently support fairness in the upper levels and fairness or FIFO in the lower levels, which matches the hierarchical policies supported by the Hadoop scheduler [6]. Determining how to extend this to other hierarchical policy sets (for example, with finish time fairness) is future work.

Gavel solves hierarchical objectives using a procedure called water filling [15], which is used in other max-min fairness problems such as link allocation in networks [49]. At a high level, the water-filling algorithm increases the allocation given to all parties at an equal rate to respect max-min fairness, until a party saturates. The saturated party is then taken out, and the procedure repeated iteratively until all commodities are saturated. We adapt this procedure to our setting, solving a series of optimization problems iteratively: an LP that computes a fair allocation across entities while respecting each entity's internal policy, and an MILP that identifies *bottlenecked jobs*, i.e., jobs whose effective throughputs cannot be improved without lowering other jobs' effective throughput.

We assume that each entity s is associated with a weight w_s; the jobs belonging to this entity receive a total cluster share proportional to this weight. We denote w_m^{job} to be the weight of job m, set such that $\sum_{m \in s} w_m^{\text{job}} = w_s$. Jobs are assigned priorities in accordance to the relevant entity's policy; for example, a fairness policy within an entity would assign each job a weight proportional to its individual weight within the entity, while for FIFO, the first job in the queue would initially receive the entire weight of the entity.

In each iteration, we solve the following modified LP (assuming $\text{scale\_factor}_m = 1$ for all m for simplicity):

$$\text{Maximize}_X \min_{\{m: w_m^{\text{job}} > 0\}} \frac{1}{w_m^{\text{job}}} \left(\frac{\text{throughput}(m, X)}{\text{throughput}(m, X_m^{\text{equal}})} - t_m \right)$$

t_m is the normalized effective throughput of job m in the previous iteration ($t_m := 0$ in the first iteration). The above objective can be appropriately modified for $\text{scale\_factor}_m > 1$. Bottlenecked jobs are given priority 0 and no longer considered in future iterations. Priorities are redistributed among non-bottlenecked jobs according to the entity's policy at the end of every iteration. For instance, in the example shown in Figure 6, if job 4 is bottlenecked, then its weight is reassigned to job 5 in accordance to the FIFO policy, while if job 2 is bottlenecked, its weight is distributed equally between jobs 1 and 3 in accordance with the entity's fairness policy. The LP then solves the max-min problem on the resources remaining while ensuring each job's throughput does not drop compared to the previous iteration's allocation X^{prev}, expressed as $\text{throughput}(m, X) \geq \text{throughput}(m, X^{\text{prev}})$ for all m. Iterations continue until all jobs are bottlenecked. To make this procedure more concrete, consider an example with 4 identical jobs: job 1 with a weight of 3.0, and jobs 2 to 4 with a weight of 1.0; and 4 identical GPUs. In the first iteration, job 1 is assigned resources such that its throughput is 1.0, and jobs 2, 3, and 4 are assigned resources such that their throughput is 0.33 to respect weights. Job 1 is a bottleneck; the throughput of the remaining jobs can still be increased. In the next iteration, jobs 2 to 4 are given full-GPU allocations.

The final allocation satisfies both inter-entity and intra-entity policies. We note that the above water-filling procedure can also be used for single-level fairness policies such as the one described in §4.1 to improve the throughput of non-bottelenecked jobs.

4.4 Properties of Gavel's Policies

Existing scheduling schemes have been analyzed in terms of properties like sharing incentive, Pareto efficiency, and strategy proofness [26]. We formalize Gavel's heterogeneity-aware policies in the context of these properties as well.

Homogeneous Clusters. For homogeneous clusters, Gavel's heterogeneity-aware policies are equivalent to the baseline policies (throughput$(m, X) = X_m \cdot T_m$), since the heterogeneity-aware optimization problems reduce to the original optimization problems with one accelerator type.

Sharing Incentive. For heterogeneous clusters, the policy's objective metric (maximize least job share in LAS, completion time of first job in FIFO, or makespan) is at least as well off as it would be under a policy that naïvely splits all resources equally among all runnable jobs. This is because the allocation corresponding to giving each user $1/n$ of each resource is a feasible solution to Gavel's optimization problem, so Gavel's solution will be at least as good. All Gavel policies have *sharing incentive* [26], which encourages users to use the shared cluster rather than a static private share.

Colocation. Solutions with colocation are always at least as good as without colocation.

Pareto Efficiency. Allocations of max-min fairness policies with water filling are Pareto efficient: that is, the allocation for a particular job cannot be increased without decreasing the allocation for another job.

Note that some of Gavel's policies may not satisfy other desirable properties. For example, Sun et al. [53] showed that no fair-sharing policy can simultaneously satisfy Pareto efficiency, sharing incentive and strategy proofness in a setting with interchangeable resources. If users manipulate their throughputs, then they can possibly obtain larger shares of the cluster (e.g., jobs can be placed on a faster accelerator type) for certain objectives. Exploring how to make Gavel's policies strategy-proof is interesting future work.

5 Scheduling Mechanism

Gavel's scheduling mechanism schedules training iterations of runnable jobs on the available workers (with possibly different accelerators), such that for each schedulable job (or combination), the fraction of wall-clock time it spends on each accelerator type is approximately equal to the computed optimal allocation X^{opt} *between* allocation recomputation events. This is challenging for two main reasons: 1) Jobs can run on multiple accelerators. Moreover, since distributed training can be communication intensive [19, 46], jobs should be placed on accelerators "close" to each other (for example, on accelerators on the same server, or on accelerators in servers in the same rack). 2) Combinations of up to two jobs can run on a set of accelerators in order to improve resource utilization (space sharing). Each distinct job can have ≤ 1 job combination running in a given round to prevent work duplication.

Gavel makes its scheduling decisions in *rounds*. This is similar in spirit to Tiresias's [28] priority discretization in some respects. However, Gavel's scheduling mechanism differs from Tiresias's in three ways:

- Gavel needs to schedule jobs on different accelerator types: it needs to decide which job should be active in any round *and* which accelerator type to use.
- Gavel needs to grant resources to jobs while respecting an *arbitrary allocation* returned by the policy.
- Gavel's round-based scheduler grants time to jobs while ensuring that multiple job combinations sharing a job do not run in the same round; Tiresias does not consider job combinations and does not need to deal with this.

Gavel's scheduler tries to place work on all available workers for a specific duration (this time period is configurable; we use 6 minutes in our experiments). We call the work handed to each worker in a given round a *micro-task*. Without rounds, jobs that request many accelerators can suffer from starvation. For example, consider a cluster with 8 total accelerators and 4 available. The scheduler can handle a 8-accelerator job waiting for resources in one of two ways: a) wait for 8 accelerators to become available; 4 accelerators will be unused until the full quota of 8 accelerators becomes available, b) keep the 8-accelerator job in the queue, and give 4 accelerators to another job that requests a fewer number of resources. However, this situation can repeat itself, leading to starvation [59]. Scheduling is thus performed in rounds to limit resource under-utilization, simplify scheduling logic, and ensure that jobs with large scale factors do not experience prolonged starvation.

Since the number of active, *schedulable* jobs might far exceed the total number of workers, Gavel first determines the job combinations that should run in the upcoming round. To do this, Gavel maintains the time t_{mj} spent by a job (or combination) m on accelerator type j, which is updated as jobs run on different accelerator types every round. Given t_{mj}, Gavel's scheduler can then compute the fraction of total wall-clock time spent by each job (or combination) m on each accelerator type j as $f_{mj} = t_{mj}/(\sum_{m'} t_{m'j})$. The matrix of priorities is then just the element-wise division of X^{opt} by f.

Algorithm. In every round, we want to move f_{mj} closer to X^{opt}_{mj}. This can be achieved by giving high-priority jobs time on accelerator type j.

This problem can be solved exactly if jobs only request single accelerators and if space sharing is not deployed by finding the num_workers$_j$ jobs with highest priority (for example, using a heap). However, jobs submitted to Gavel can be distributed, and space sharing can be used to improve resource utilization. Solving this problem exactly with these added requirements makes the problem similar to a multiple-choice knapsack problem [52], which is NP-hard.

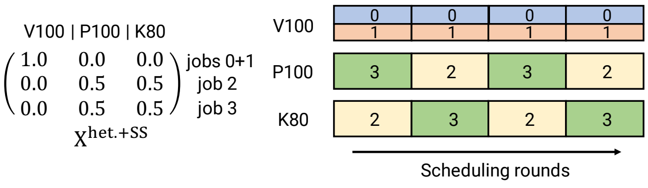


Figure 7: Round-based scheduling mechanism in action to achieve an allocation $X^{\text{het.+SS}}$. Space sharing is shown with vertically split boxes. Each round is denoted by a box.

Algorithm 1 Algorithm for Gavel's scheduling mechanism

```
function SCHEDULE_JOBS
    active_combinations ← all active job combinations
    num_workers_rem. ← number of total workers
    while num_workers_rem.g > 0 do
        $j$ ← job combination with highest priority
        Remove $j$ from active_combinations
        if $j$.scale_factor > num_workers_rem. then
            continue
        for all $j'$ that conflict (share a job $k$) with $j$ do
            Remove $j'$ from active_combinations
        num_workers_rem. −= $j$.scale_factor
```

To overcome these challenges, we observe that it is acceptable to make greedy sub-optimal *scheduling* decisions occasionally in any given round, since we can recover from these sub-optimal decisions in subsequent rounds: our goal is to ensure that the average allocation each job receives *over multiple rounds* resemble the computed allocation (the allocations returned by policies are optimal, which follows from how policies in Gavel are expressed as optimization problems). We study the impact of this design choice in §7.5. A job (combination) not run in a particular round will have increased priority in subsequent rounds until it receives accelerator time, while a job that runs in a particular round will have decreased priority. This ensures that jobs do not suffer from starvation if they have a non-zero optimal allocation.

Gavel uses a greedy algorithm to pick the highest-priority job combinations that fit in the provided resource budget. The algorithm maintains a set of eligible job combinations (`eligible_job_combinations`) that can be scheduled in the upcoming scheduling round. The scheduling mechanism then tries to add job combinations with highest priority into a `job_combinations_to_schedule` set. Once a job combination is added to this set, all *conflicting* job combinations are removed from the set of eligible combinations to ensure that a given job is not run more than once in a given scheduling round. Job combinations that cannot fit in the current round due to space limitations (required number of accelerators unavailable) are also removed from the set of eligible combinations. This procedure is detailed in Algorithm 1. Gavel's scheduling mechanism is decoupled from its policies, ensuring that the same scheduling mechanism can be used for many different policies. Figure 7 shows Gavel's scheduling mechanism in action.

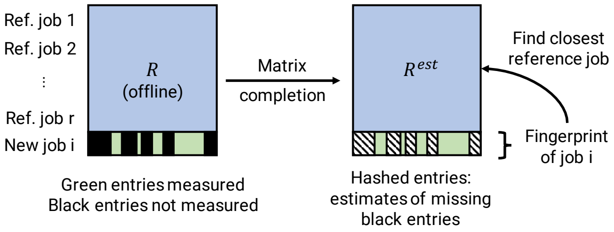


Figure 8: Gavel's throughput estimator. Profiling is combined with matrix completion to obtain a fingerprint for every new job. The fingerprint is then used to find the closest reference job.

Once Gavel has decided what jobs (and combinations) should run in a given round on different accelerator types, Gavel must decide how to *place* these jobs. Gavel's scheduler places jobs in decreasing order of the number of requested workers, and tries to give jobs accelerators on the same physical server to minimize fragmentation.

6 Implementation

We implemented a prototype of Gavel in approximately 9000 lines of Python code, and implemented a simulator in about 500 LOC. We used `cvxpy` [23] to implement Gavel's heterogeneity-aware policies, and `gRPC` [4] to communicate control messages between the scheduler and workers.

Interface between Scheduler and Applications. Gavel currently supports user applications written in PyTorch [48]; support for TensorFlow [13] is left for future work. The scheduler and user applications then interact through a narrow API. Gavel ships with a Python library that users can import into their code. This library provides an implementation for a wrapper around existing framework-provided data iterators (`GavelIterator`). `GavelIterator` ensures that each task in a distributed job runs for the same number of iterations, and synchronizes the conclusion of rounds between the scheduler and workers. `GavelIterator` is instantiated with arguments `train_loader` (base data loader), `load_checkpoint`, `save_checkpoint`, and a configuration object. `load_checkpoint` is a pointer to a function that loads all necessary parameters and metadata from a checkpoint at the start of a round, and `save_checkpoint` is a pointer to a function that creates a checkpoint at the end of a round; these need to call appropriate framework methods ($<$ 5 LOC).

`GavelIterator` contacts the scheduler near a round end to see if the same job will run in the next round on the same worker. We call this a *lease renewal*. If the lease is not renewed, the iterator calls `save_checkpoint` at round end. The scheduler can then launch another job on the worker.

Throughput Estimation. Gavel uses a similar technique to Quasar [21] to estimate colocated throughputs when using the optional space sharing optimization (if they are not available a priori), mixing profiling with matrix completion.

Model	Task	Dataset / Application	Batch size(s)
ResNet-50 [5, 31]	Image Classification	ImageNet [22]	16, 32, 64, 128
ResNet-18 [31, 39]	Image Classification	CIFAR-10 [36]	16, 32, 64, 128, 256
A3C [27, 44]	Deep RL	Pong	4
LSTM [11]	Language Modeling	Wikitext-2 [42]	5, 10, 20, 40, 80
Transformer [33, 55]	Language Translation	Multi30k [24] (de-en)	16, 32, 64, 128, 256
CycleGAN [38, 60]	Image-to-Image Translation	monet2photo [60]	1
Recoder [45] (Autoencoder)	Recommendation	ML-20M [29]	512, 1024, 2048, 4096, 8192

Table 2: Models used in the evaluation.

Trace	System	Objective	Physical	Simulation
Continuous	Gavel	Average JCT	3.4 hrs	3.7 hrs
Continuous	LAS	Average JCT	5.1 hrs	5.4 hrs
Static	Gavel	Makespan	17.7 hrs	17.6 hrs
Static	Gandiva	Makespan	21.3 hrs	22.1 hrs

Table 3: Comparison of end objective between physical experiment and simulation for two different traces. For the continuous trace, we measure the average JCT of 25 jobs in a steady-state cluster. For the static trace, we measure the total time needed to complete 100 jobs submitted at the start of the run. The heterogeneity-aware policies improve target objectives, and results on the physical cluster are in agreement with results on simulated cluster (< 8%).

Model	Overhead without lease renewals	Overhead with lease renewals
ResNet-18	0.94%	0.17%
ResNet-50	1.58%	0.25%
A3C	0.22%	0%
LSTM	2.91%	0.47%
Transformer	0.77%	0.11%
CycleGAN	0.77%	0.11%

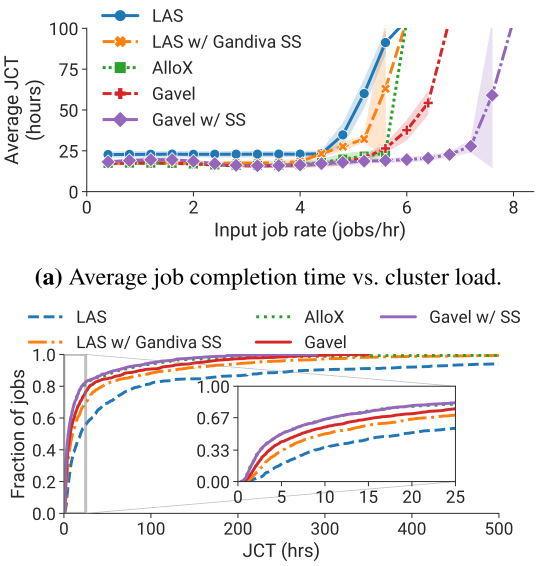
Table 4: Overhead of using preemptive scheduling in Gavel, with and without lease renewals, and with a round duration of 6 minutes.

Matrix completion enables sparse low rank matrices to be reconstructed with low error [17, 43]. With matrix completion, Gavel is able to extrapolate measurements obtained through direct profiling on separate workers dedicated to profiling, and determine the job's most similar pre-profiled reference job. The throughput estimator can then use the reference job's throughput measurements as an initial throughput estimate. Gavel's throughput estimator is diagrammed in Figure 8.

7 Evaluation

In this section, we seek to answer the following questions:

- Do Gavel's heterogeneity-aware policies improve objective metrics in a physical cluster (§7.2) and in simulations of larger clusters (§7.3)?
- How do Gavel's policies scale? (§7.4)



(a) Average job completion time vs. cluster load.

(b) CDF of job completion times (input job rate = 5.6 jobs/hr).

Figure 9: Comparison of heterogeneity-agnostic least attained service (LAS) policy to a heterogeneity-aware LAS policy (Gavel), in simulation on the continuous-single trace.

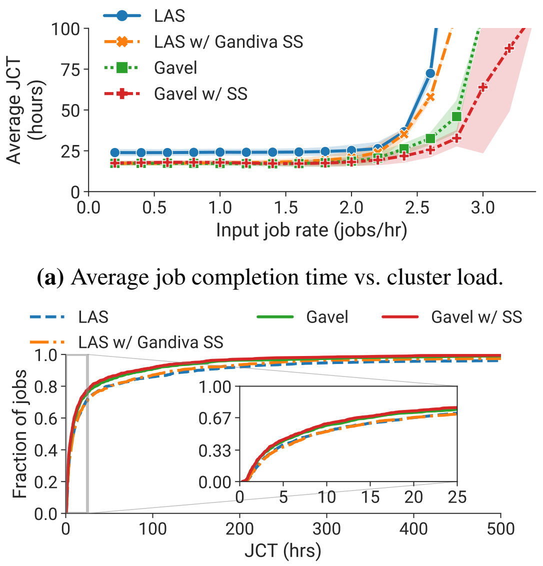
- How well does Gavel's scheduling mechanism realize Gavel's heterogeneity-aware allocations? (§7.5)
- Is Gavel able to accurately estimate the throughputs of co-located jobs when using space sharing? (§7.6)

7.1 Experiment Setup

We run experiments on both a physical and simulated cluster.

Clusters. We run physical cluster experiments on a cluster with 8 V100s, 16 P100s, and 24 K80s. Simulated cluster experiments are run on a cluster with 36 GPUs of each type.

Traces. We run physical and simulated experiments on two types of traces: one where all jobs are available at the start of the trace and jobs are *not* subsequently added ("static"), and another where jobs are continuously added to the cluster ("continuous"). For the continuous trace, job arrival times are generated according to a Poisson arrival process with an inter-arrival rate λ. For the simulated experiments, we vary λ to show the extra load each heterogeneity-aware policy is able to sustain in steady state. We run 3 seeds for every λ, and show standard deviations. For the physical cluster experiments, we use a single λ that keeps the cluster well-utilized in steady state. The online traces used in the simulated experiments have a variable number of jobs (at least 5000) and span 20-30 days. We measure the completion times of jobs with ID 4000 to 5000 to study steady state behavior (new jobs continue to be added until jobs of interest complete). Job types are uniformly sampled from the job table with 26 distinct job (or model) types, shown in Table 2. The online traces used in the physical experiments span a day and have 100 jobs.



(a) Average job completion time vs. cluster load.

(b) CDF of job completion times (input job rate = 2.6 jobs/hr).

Figure 10: Comparison of heterogeneity-agnostic least attained service (LAS) policy to a heterogeneity-aware LAS policy (Gavel), in simulation on the continuous-multiple trace. Each input job rate is run with 3 seeds; shaded regions show the standard deviation.

The duration of each job *on a V100 GPU* is sampled from an exponential distribution: jobs have duration 10^x minutes, where x is drawn uniformly from $[1.5,3]$ with 80% probability, and from $[3,4]$ with 20% probability. Given the job's observed throughput on the V100 GPU, the number of training steps is then inferred by multiplying the throughput (in steps/sec) by the duration. This matches the process used by Gandiva [58]. For the simulated experiments, we show results in two regimes: one where all jobs use a single worker ("continuous-single"), and another where 70% of jobs request a single worker, another 25% request between 2 and 4 workers, and the remaining 5% request 8 workers, as observed in published traces from Microsoft [9] ("continuous-multiple").

Metrics. For fairness and FIFO policies, our target metric is average job completion time of steady-state jobs, which is the same metric used by related work [28, 41]. We also show finish time fairness (FTF) for policies that explicitly optimize for FTF. For makespan policies, our target metric is the time needed to complete a job batch. For cost-related policies, the metric is cost (in dollars), and the percentage of jobs that violate time SLOs.

7.2 End-to-End Results on Physical Cluster

For our physical cluster experiments, we run a heterogeneity-aware and a heterogeneity-agnostic fairness policy on a continuous trace, and a heterogeneity-aware makespan policy against a baseline that uses Gandiva's ad-hoc space sharing on a static trace. Results are shown in Table 3. Gavel's heterogeneity-aware policies improved average job completion time by **1.5**× and makespan by **1.2**×. For the makespan objective, we do not run Gavel with space sharing; in theory, space sharing would additionally reduce makespan.

We also compare the real performance to simulations and observe that for both policies, the difference between metrics in simulation and on the physical cluster is small ($< 8\%$), indicating that our simulator has high fidelity.

Table 4 shows the overhead of using Gavel's preemptive scheduler with a round duration of 6 minutes, with and without lease renewals. Allocations and worker assignments can be computed asynchronously. The only synchronous overhead is the loading and saving of checkpoints, which is dependent on the size of the model. Lease renewals decrease this overhead by allowing jobs to run on the same worker for extra rounds. The overhead of preemption, even without lease renewals and with a short round duration, is low ($< 3\%$).

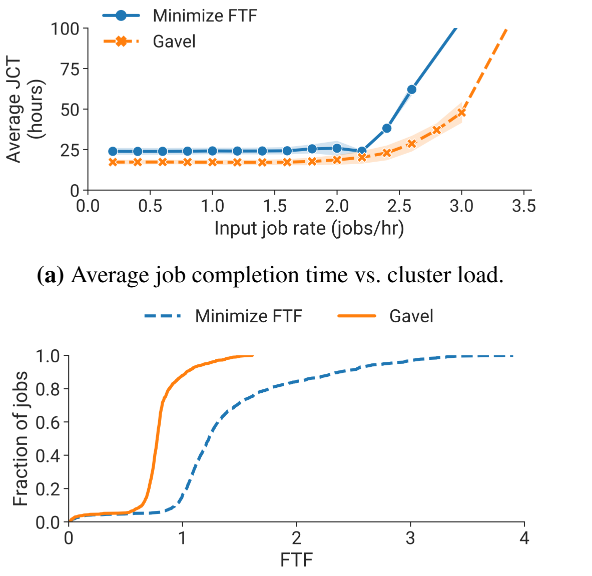
7.3 End-to-End Results in Simulation

We use a larger simulated cluster to evaluate the efficacy of Gavel's heterogeneity-aware policies across a range of objectives, and compare with heterogeneity-agnostic versions from previous work using a round duration of 6 minutes. As appropriate, we compare to other baselines like AlloX. Magnitudes of speedups are higher for these experiments compared to the physical cluster experiments since the simulated traces show job behavior over weeks, while the physical cluster traces are only a day long; consequently, queue buildups are less extreme for the traces used in the physical cluster experiments.

Least Attained Service (LAS). Figures 9 and 10 compare the vanilla LAS policy with its heterogeneity-aware variants. We compare with two other baselines: a modified LAS policy that uses Gandiva's ad-hoc space sharing, and an AlloX policy that explicitly optimizes average job completion time (but only for single-worker jobs). We make three observations.

First, the heterogeneity-aware policies support higher load on the *same* cluster, reduce average JCT by **3.5**× for the continuous-single trace, and by **2.2**× for the continuous-multiple trace (graph can be read by comparing average JCT value for a given input job rate or x-intercept) at high load (5.6 jobs/hr for continuous-single, 2.6 jobs/hr for continuous-multiple). Second, the heterogeneity-aware LAS policy supports higher load than AlloX, since AlloX can give short jobs preferential treatment in the interest of optimizing average JCT, leading to long jobs experiencing starvation (long tail in JCT CDF). At moderate load, AlloX represents a best-case scenario since it explicitly optimizes for average JCT on a heterogeneous cluster. Gavel is able to essentially match this best case scenario, while also supporting other objectives. Third, Gandiva-style packing, which randomly explores job combinations until a combination that improves performance is found, is ineffective compared to Gavel's principled packing (**2.2**× better average JCT for both traces at high load).

Finish Time Fairness (FTF). We compare the heterogeneity-aware version of Finish Time Fairness



(a) Average job completion time vs. cluster load.

(b) CDF of finish time fairness metric (input job rate = 2.6 jobs/hr).

Figure 11: Comparison of a heterogeneity-agnostic policy that optimizes for finish time fairness ("Minimize FTF") to a heterogeneity-aware one (Gavel), in simulation with the continuous-multiple trace.

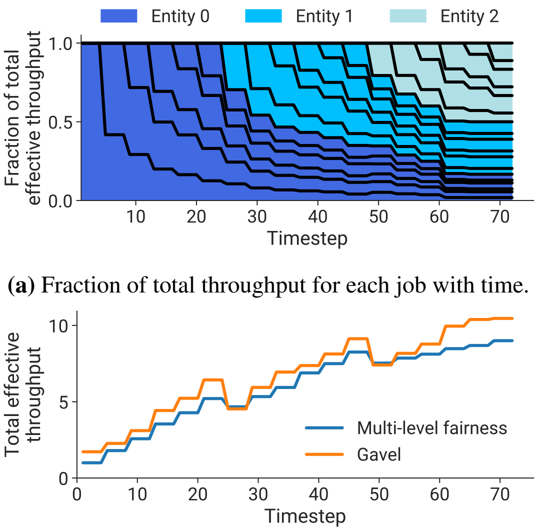
(FTF) to its heterogeneity-agnostic counterpart in Figure 11. The heterogeneity-aware policy reduces average JCTs by **3**× and improves average FTF by **2.8**×. FTF is the ratio of the time taken to finish a job using a given allocation and the time taken to finish the job using $1/n$ of the cluster (X^{isolated}), assuming n users use the cluster. Lower FTF means jobs take less time with the provided allocation compared to X^{isolated}.

Makespan. Gavel's heterogeneity-aware makespan policy reduces makespan by **2.5**× compared to a FIFO baseline, and by **1.4**× compared to a baseline that uses Gandiva's ad-hoc space sharing. Makespan is reduced by a further **8%** when the number of jobs in the trace is high when using space sharing.

FIFO. The heterogeneity-aware versions of FIFO allow the cluster to support average input job rate. At high load, the heterogeneity-aware version without space sharing reduces average JCT by **2.7**×, and the heterogeneity-aware version with space sharing reduces average JCT by **3.8**× at high load. Space sharing is less effective for distributed jobs: it reduces average JCT by **1.1**× with distributed jobs, compared to **1.4**× for the continuous-single trace.

LAS with priorities. We also run an experiment with the LAS policies where 20% of jobs have higher priority. At high load, Gavel reduces the average JCT of high-priority jobs by **1.5**× and the average JCT of low-priority jobs by **2.7**×.

Cost. We simulate each of the cost policies on a 500-job workload comprised of ResNet-50 and A3C jobs. As we observe in Figure 1b, the ResNet-50 job has the best cost-normalized throughput on the V100 while the A3C job has the best cost-normalized throughput on the K80. Each job's duration is chosen from $\{0.5, 1, 2, 4, 8\}$ days, and each job's SLO is chosen from $\{1.2\times, 2\times, 10\times\}$ its duration.



(a) Fraction of total throughput for each job with time.

(b) Total throughput vs. time.

Figure 12: Behavior of a multi-level fairness policy with time as jobs are added to a small cluster with 3 V100 GPUs, 3 P100 GPUs, and 3 K80 GPUs. Each line represents a separate job, and jobs are added every 4 timesteps. The first 6 jobs belong to entity 0 (weight of entity, $w_0 = 1$), the next 6 jobs belong to entity 1 ($w_1 = 2$), and the last 6 jobs belong to entity 2 ($w_2 = 3$).

The policy that minimizes cost reduces the total cost compared to the policy that maximizes throughput by a factor of roughly **1.4**×. However, approximately **35%** of jobs violate their SLO as this policy prioritizes cheaper but slower GPUs; in particular, the A3C jobs are scheduled on K80 GPUs which results in violations for tight SLOs. In comparison, the policy that includes SLOs as well eliminates all violations for a small increase in cost (a cost reduction of **1.2**× compared to the baseline policy), by ensuring that A3C jobs with tight SLOs are run on instances with V100 GPUs.

Multi-level Hierarchical Policies. Figure 12 shows the behavior of a multi-level fairness policy as new jobs belonging to multiple entities are added to a heterogeneous cluster with equal numbers of K80, P100, and V100 GPUs. Resources are granted to jobs in a way that respects both the higher-level and lower-level policies: in Figure 12a, fairness is enforced both within and across entities (as can be seen by the widths of the colored bands, which represents cross-entity fairness, and the widths of bands within a color, which represents fairness across jobs within an entity), and allocations are adjusted as new jobs come in. Figure 13 shows results with a fairness+FIFO policy; later jobs in each entity 0 do not receive any GPU time to respect the per-entity FIFO policy.

The multi-level fairness policy can also be implemented in a heterogeneity-agnostic manner by statically partitioning

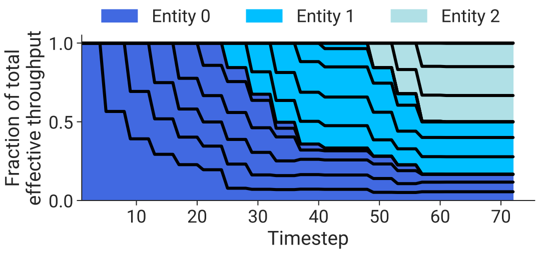
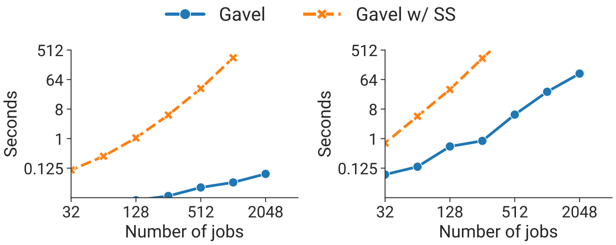


Figure 13: Behavior of a hierarchical policy (weighted fairness as top-level policy, FIFO as bottom-level policy) with time as jobs are added to a small cluster with 3 V100 GPUs, 3 P100 GPUs, and 3 K80 GPUs. Each line represents a separate job, and jobs are added every 4 timesteps. The first 6 jobs belong to entity 0 (weight of entity, $w_0 = 1$), the next 6 jobs belong to entity 1 ($w_1 = 2$), and the last 6 jobs belong to entity 2 ($w_2 = 3$).



(a) LAS.

(b) Hierarchical.

Figure 14: Scaling of LAS and hierarchical policies with the number of active jobs on a heterogeneous cluster with an equal number of V100, P100, and K80 GPUs. The size of the cluster is increased as the number of active jobs is increased.

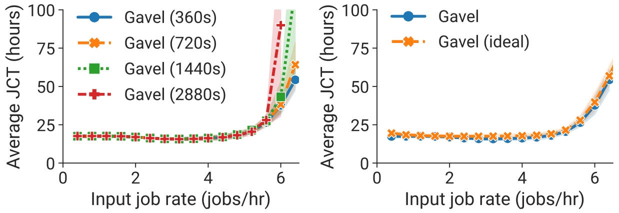
resources across users while respecting per-entity and per-user weights. While this results in a fair allocation as well, we observe that total effective throughput is about **17%** lower compared to the heterogeneity-aware policy (Figure 12b).

7.4 Scalability of Heterogeneity-Aware Policies

Figure 14 shows the scaling behavior of the heterogeneity-aware LAS and multi-level fairness policies with and without space sharing. We observe that even with 2048 active jobs, the hierarchical policy without space sharing can be run in $<$ 10 minutes. With space sharing, the policy can be run with 512 jobs in $<$ 10 minutes. The single-level LAS policy is much cheaper to compute in comparison. We note that allocations do not need to be recomputed every scheduling round – however, the longer the policy takes to run, the longer it takes for the new allocation to be acted upon (jobs can still be given heterogeneity-agnostic allocations in the interim, and consequently time on resources). We believe latencies of $<$ 30 minutes for large clusters are still preferable to non-preemptive schedulers where jobs experience large queuing delays, or preemptive schedulers with heterogeneity-agnostic policies which lead to worse objective values, as shown above.

7.5 Efficacy of Scheduling Mechanism

Figure 15a shows the effect of the round length on average JCT for the heterogeneity-aware LAS policy with a single-GPU trace. We observed similar behavior on traces with multi-GPU jobs, as well as other policies. A smaller round length gives Gavel's scheduling mechanism more rounds to course correct, allowing the true allocation and computed optimal allocation to more closely match. We found that the time needed to load and save checkpoints for our target models is $<$ 5 seconds, which means that a round length of 6 minutes gives a good tradeoff between fidelity with the optimal allocation and preemption overhead (preemption overhead with 6-minute rounds shown in Table 4).



(a) Effect of round length.

(b) Mechanism vs. ideal.

Figure 15: (a) Effect of round length on average JCT for the heterogeneity-aware LAS policy. (b) Comparison of scheduling mechanism to an ideal baseline that allocates resources to jobs *exactly* according to the computed allocation for the same policy.

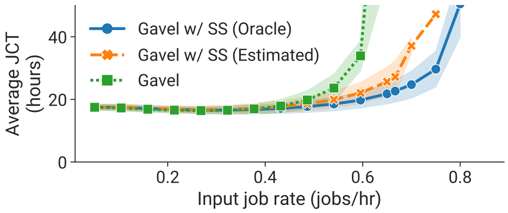


Figure 16: Comparison of SS-aware LAS policy with estimated throughputs, compared to the SS-aware with oracle throughputs and LAS without space sharing on a heterogeneous 12-GPU cluster.

We compare this to an ideal baseline that allocates resources to jobs *exactly* according to their computed allocation. As shown in Figure 15b, Gavel's scheduling mechanism with a round duration of 6 minutes behaves almost identically to this ideal baseline with a single-GPU trace (behavior with a multi-GPU trace is similar). We note that the ideal baseline is impractical to use in practice, since jobs with different scale factors can complete at different times (leading to starvation), and preemptions can be often since allocations for some (job, accelerator type) pairs are small, leading to high overhead.

7.6 Impact of Throughput Estimation

Figure 16 shows the effect of Gavel's throughput estimator on average JCT when using the space sharing-aware LAS policy compared to the LAS policy without space sharing, and the LAS policy with space sharing and oracle throughputs. The throughput estimator is able to determine missing throughputs in an online fashion accurately enough to observe a very small decrease in average JCT at high load (orange and blue lines).

8 Related Work and Discussion

In this section, we compare Gavel to related work.

Existing DNN Training Schedulers. Several recent papers have proposed schedulers targeting DNN training workloads.

Gandiva [58] uses time and space sharing to reduce queuing delay and improve resource utilization, but does not specify an explicit scheduling policy and does not support configurable objectives. It uses a profiling-based methodology to determine whether to co-locate jobs on an accelerator. However, it does not incorporate model performance data (isolated or co-located performance) explicitly into its scheduling policy, resorting to random exploration of job combinations until a combination that improves performance is found.

Tiresias [28] and Themis [40] use different objectives to achieve multi-job fairness. However, both do not incorporate jobs' affinities for different accelerator types in their scheduling objectives, and have scheduling mechanisms strongly coupled with the target policy, making it hard to support other more sophisticated policies like multi-level fairness.

AlloX [37] and Gandiva$_{\text{fair}}$ [18] are recent DNN schedulers that do consider worker and model heterogeneity. However, both only work for single policies (average job completion time for AlloX, max-min fairness for Gandiva$_{\text{fair}}$). Moreover, Gandiva$_{\text{fair}}$ uses a second-price auction mechanism to improve the performance of a heterogeneity-agnostic max-min fairness scheme, but does not provide guarantees as to the optimality of the final allocation. On the other hand, Gavel formalizes each policy as an optimization problem, and can provide a guarantee that the returned solution is "optimal" according to the provided objective. Gavel is also able to support more sophisticated policies such as multi-level fairness.

Traditional Cluster Schedulers. Traditional schedulers such as Mesos [32], Borg [57], TetriSched [54], and YARN [56] support workloads with fixed heterogeneous resource requests, but do not reason about the diverse performance characteristics of jobs across accelerators. Mesos and YARN do not reason about interchangeable resource types that can run the same computation: for example, Mesos's DRF multi-resource sharing policy [26] decides how to give jobs allocations of distinct resource types, such as RAM and CPUs, but assumes that each job has declared which resources it needs to use and in what ratio (unlike our case, where we consider heterogeneity over accelerators themselves).

The multi-interchangeable resource allocation (MIRA) problem [53] also introduces the notion of effective throughput similar to Gavel, but does not demonstrate how this can be used to specify policies as optimization problems, does not consider performance optimizations like space sharing and placement sensitivity, and does not discuss how computed allocations can be realized on physical resources.

Omega [50], Apollo [16], and Hydra [20] are schedulers that take into account the fact that the target workload shows heterogeneity in the number and duration of constituent tasks. However, tasks largely take the same time on different CPUs, and heterogeneity in memory capacities only impacts the number and size of tasks that can be placed on a server. In our work, the compute devices themselves are interchangeable with sometimes large performance differences, and policies decide the time fractions of resources each job should receive while optimizing for various end objectives.

Dynamic Performance Estimation. As detailed in §6, Gavel uses the approach proposed by Quasar [21] to estimate co-located job performance online. In particular, Gavel uses a mix of profiling and matrix completion to compute a "fingerprint" against a set of reference models profiled offline. In this work, we show that the techniques used by Quasar can be successfully applied to this new setting.

Applicability to Other Settings. Even though we focused this paper on allocating heterogeneous resources for DNN training workloads, we believe that Gavel can be used for non-DNN workloads as well. Other workloads that are amenable to GPU execution, such as simulations, can be considered, even though performance estimates for these applications will be needed. We also believe the main technical insight presented in this paper – formulating diverse scheduling policies as optimization problems – is broadly applicable, and can be used to more easily deploy policies on homogeneous deep learning clusters, and on CPU clusters as well.

9 Conclusion

In this paper, we proposed Gavel, a heterogeneity-aware cluster scheduler that is able to optimize for many high-level metrics like fairness, makespan, and cost. Gavel demonstrates how existing policies can be expressed as optimization problems, and extends these policies to be heterogeneity-aware. Gavel then uses a decoupled round-based scheduling mechanism to ensure that the computed optimal allocation is realized. Gavel's heterogeneity-aware policies improve end objectives both on a physical and simulated cluster. It can support a higher average input job rate, while improving objectives such as average job completion time by 3.5×, makespan by 2.5×, and cost by 1.4×.

10 Acknowledgements

We thank our shepherd, Alexandra Fedorova, the anonymous OSDI reviewers, Firas Abuzaid, Trevor Gale, Shoumik Palkar, Deepti Raghavan, Daniel Kang, Pratiksha Thaker, and fellow Project Fiddle interns Jack Kosaian, Kshiteej Mahajan, and Jayashree Mohan for their invaluable feedback that made this work better. We thank MSR for their generous support of DN's and KS's internships, and for resources to develop and evaluate Gavel. This research was also supported in part by affiliate members and other supporters of the Stanford DAWN project— Ant Financial, Facebook, Google, Infosys, NEC, and VMware—as well as Toyota Research Institute, Northrop Grumman, Amazon Web Services, Cisco, NSF Graduate Research Fellowship grant DGE-1656518, and the NSF 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.

References

[1] AWS Accelerator Offerings. https://aws.amazon.com/ec2/instance-types/, 2020.

[2] Cloud GPUs on GCP. https://cloud.google.com/gpu, 2020.

[3] Cloud TPUs on GCP. https://cloud.google.com/tpu, 2020.

[4] gRPC. https://grpc.io, 2020.

[5] ImageNet Training in PyTorch. https://github.com/pytorch/examples/tree/master/imagenet, 2020.

[6] Implementing Core Scheduler Functionality in Resource Manager (V1) for Hadoop. https://issues.apache.org/jira/browse/HADOOP-3445, 2020.

[7] Job Scheduling in Spark. https://spark.apache.org/docs/latest/job-scheduling.html#scheduling-within-an-application, 2020.

[8] Linear-fractional Optimization. http://www.seas.ucla.edu/~vandenbe/ee236a/lectures/lfp.pdf, 2020.

[9] Microsoft Philly Trace. https://github.com/msr-fiddle/philly-traces, 2020.

[10] NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2020.

[11] Word-level Language Modeling RNN. https://github.com/pytorch/examples/tree/master/word_language_model, 2020.

[12] YARN – The Capacity Scheduler. https://blog.cloudera.com/yarn-capacity-scheduler/, 2020.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. 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.

[14] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[15] D. P. Bertsekas and R. G. Gallager. Data Networks. 1987.

[16] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, 2014.

[17] E. J. Candes and Y. Plan. Matrix Completion with Noise. *Proceedings of the IEEE*, 98(6):925–936, 2010.

[18] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[19] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. Analysis of DAWNBench, A Time-to-Accuracy Machine Learning Performance Benchmark. *ACM SIGOPS Operating Systems Review*, 53(1):14–25, 2019.

[20] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, et al. Hydra: A Federated Resource Manager for Data-Center Scale Analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 177–192, 2019.

[21] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 127–144, 2014.

[22] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[23] S. Diamond and S. Boyd. CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016.

[24] D. Elliott, S. Frank, K. Sima'an, and L. Specia. Multi30K: Multilingual English-German Image Descriptions. In *Proceedings of the 5th Workshop on Vision and Language*, pages 70–74. Association for Computational Linguistics, 2016.

[25] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, et al. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[26] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, pages 24–24, 2011.

[27] D. Griffis. RL A3C PyTorch. https://github.com/dgriff777/rl_a3c_pytorch, 2020.

[28] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[29] F. M. Harper and J. A. Konstan. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems (TIIS)*, 5(4):19, 2016.

[30] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2961–2969, 2017.

[31] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[32] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. 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)*, pages 22–22, 2011.

[33] Y.-H. Huang. Attention is All You Need: A PyTorch Implementation. https://github.com/jadore801120/attention-is-all-you-need-pytorch, 2018.

[34] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *USENIX Annual Technical Conference, USENIX ATC 2019*, pages 947–960, 2019.

[35] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[36] A. Krizhevsky, V. Nair, and G. Hinton. The CIFAR-10 Dataset. http://www.cs.toronto.edu/kriz/cifar.html, 2014.

[37] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu. AlloX: Compute Allocation in Hybrid Clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[38] E. Linder-Norén. PyTorch-GAN. https://github.com/eriklindernoren/PyTorch-GAN#cyclegan, 2020.

[39] K. Liu. Train CIFAR-10 with PyTorch. https://github.com/kuangliu/pytorch-cifar, 2020.

[40] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[41] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. 2019.

[42] S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer Sentinel Mixture Models. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[43] A. Mnih and R. R. Salakhutdinov. Probabilistic Matrix Factorization. In *Advances in Neural Information Processing Systems*, pages 1257–1264, 2008.

[44] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[45] A. Moussawi. Towards Large Scale Training of Autoencoders for Collaborative Filtering. In *Proceedings of Late-Breaking Results Track Part of the Twelfth ACM Conference on Recommender Systems*, RecSys'18, Vancouver, BC, Canada, 2018.

[46] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[47] D. Narayanan, K. Santhanam, A. Phanishayee, and M. Zaharia. Accelerating Deep Learning Workloads through Efficient Multi-Model Execution. In *NeurIPS Workshop on Systems for Machine Learning (December 2018)*, 2018.

[48] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[49] B. Radunovic and J.-Y. Le Boudec. A Unified Framework for Max-Min and Min-Max Fairness with Applications. *IEEE/ACM Transactions on Networking*, 15(5):1073–1083, 2007.

[50] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364, 2013.

[51] M. J. Shafiee, B. Chywl, F. Li, and A. Wong. Fast YOLO: A Fast You Only Look Once System for Real-Time Embedded Object Detection in Video. *arXiv preprint arXiv:1709.05943*, 2017.

[52] P. Sinha and A. A. Zoltners. The Multiple-Choice Knapsack Problem. *Operations Research*, 27(3):503–515, 1979.

[53] X. Sun, T. N. Le, M. Chowdhury, and Z. Liu. Fair Allocation of Heterogeneous and Interchangeable Resources. *ACM SIGMETRICS Performance Evaluation Review*, 46(2):21–23, 2019.

[54] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 35. ACM, 2016.

[55] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is All You Need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[56] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[57] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18, 2015.

[58] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[59] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. 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. ACM, 2010.

[60] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2223–2232, 2017.

A Artifact Appendix

A.1 Abstract

Gavel is open sourced at https://github.com/stanford-futuredata/gavel. We provide implementations for Gavel's heterogeneity-aware policies, its round-based scheduling mechanism, and the **GavelIterator** interface, as well as implementations of relevant baselines such as AlloX [37], a simulator, and code to reproduce the graphs and other quantitative results shown in this paper.

A.2 Artifact check-list

- **Algorithm:** Heterogeneity-aware policies are expressed as optimization problems over allocations. Scheduling is performed using a greedy round-based scheduling mechanism.
- **Hardware:** Experiments in simulation can run on a multi-core server with Ubuntu 16.04. Experiments on a physical cluster need Nvidia GPUs.
- **Setup instructions:** Setup instructions are available in the README.md and EXPERIMENTS.md files provided in the artifact.
- **Experiments:** All results presented in this paper can be reproduced using the provided artifact.
- **Required disk space:** About 100 GB for logfiles when running simulated cluster experiments, about 10 GB for intermediate model checkpoints for physical cluster experiments, about 150 GB for datasets.
- **Expected experiment run time:** Days to a week for full simulated experiments, shorter durations (hours to a day) for scaled-down experiments (smaller cluster and trace).
- **Public link:** https://github.com/stanford-futuredata/gavel.
- **Code licenses:** MIT License.

A.3 Description

A.3.1 How to access

The artifact is publicly available at https://github.com/stanford-futuredata/gavel.

A.3.2 Hardware dependencies

Simulated experiments can be run on any multicore server. We ran experiments on a 56-core server with Ubuntu 16.04. Physical clusters need to have Nvidia GPU accelerators; other accelerators supported by Deep Learning frameworks such as PyTorch are supported as well by the scheduler.

A.3.3 Software dependencies

Software dependencies are specified at https://github.com/stanford-futuredata/gavel/blob/master/README.md.

A.3.4 Datasets

Running the simulator does not require any external datasets. When running physical cluster experiments, training data for training jobs is needed. These are task-specific (for example, image classification training jobs might use the ImageNet dataset).

A.4 Installation

Installation instructions are specified at https://github.com/stanford-futuredata/gavel/blob/master/README.md.

A.5 Experiment workflow

Experiments in simulation are triggered by a driver script that instantiates the scheduler, and then adds jobs to the simulated cluster either according to a pre-defined trace, or on-the-fly using distributions with input parameters specified by the user. The scheduler computes the optimal allocation for each active job based on the desired policy and target objective, and then assigns resources to jobs according to this computed allocation using its round-based scheduling mechanism. Oracle throughputs are used to estimate the progress of jobs given a specified amount of time on the given resources. At the end of a run, completion times of all jobs of interest are recorded. Jobs of interest are usually a subset of all jobs submitted to the cluster, since we want to study steady state behavior. An exception is made for makespan policies, which try to minimize the total time taken by a collection of jobs; for this policy, jobs are added once at the start of the trace, and then jobs are allowed to drain from the cluster.

Experiments on physical clusters are also triggered by a driver script run on the scheduler, but are different in one key aspect: jobs are run on real accelerators for the specified number of steps. Every round, the scheduler makes a scheduling decision to decide what resources should be given to the different jobs. As before, job completion times are recorded when a job finishes executing.

A.6 Evaluation and expected result

Each experiment run results in an output logfile that records the microtasks run every scheduling round, as well as the completion times for each job. These logfiles can then be parsed to produce the graphs and other quantitative results presented in the evaluation section of this paper. Code to parse and produce plots are available at https://github.com/stanford-futuredata/gavel/tree/master/scheduler/notebooks/figures.

A.7 Experiment customization

Experiments can be run with different seeds using the main sweep scripts. Experiments can also be scaled down in different ways to obtain results faster: a) smaller cluster, b) fewer traces, c) smaller traces, and d) smaller set of jobs of interest over which objectives (such as average JCT) are measured.

A.8 AE Methodology

Submission, reviewing and badging methodology is specified at https://www.usenix.org/conference/osdi20/call-for-artifacts.