Scheduler-driven Strategies for Fair and Efficient Data Analytics Systems

By
Kshiteej Mahajan

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the
University of Wisconsin-Madison
2021

Date of final oral examination: September 23, 2021

This dissertation is approved by the following members of the Final Oral Committee:
Aditya Akella, Professor, Computer Sciences
Shuchi Chawla, Professor, Computer Sciences
Shivaram Venkataraman, Assistant Professor, Computer Sciences
Dimitris Papailiopoulos, Assistant Professor, Electrical and Computer Engineering

I would like to dedicate this thesis to my family and all my mentors.

Acknowledgements

I started my PhD with a lack of clarity about several aspects of research. I am grateful to my advisor, Aditya Akella, for his mentorship, giving meaning to my career and nudging me towards clarity.

I am also grateful to my colleagues and collaborators for sharing their perspectives and helping me improve. Every interaction during my research career, however fleeting, has shaped me. I am, in part, a blend of all these perspectives.

I am thankful for finding amazing academic collaborators in Shuchi Chawla, Mosharaf Chowdhury, Shivaram Venkataraman, Michael Swift, Somesh Jha, Dimitris Papailiopoulos who helped me shape research ideas into research papers and helped shape this thesis.

I was fortunate to get an opportunity to collaborate with researchers at HP Labs, Microsoft Research, Facebook AI Research, Harvad, MIT and I would like to thank Sujata Banerjee, Joon-Myung Kang, Ganesh Ananthanarayanan, Yuchen Jin, Venkat Padmanabhan, Sundararajan Renganathan, Amar Phanishayee, Andrew Or, Jayashree Mohan, Srinivas Sridharan, Pallab Bhattacharya, Ching-Hsiang Chu, Minlan Yu, Mohammad Alizadeh, Hongzi Mao for making me realize how to be a better collaborator. I would like to thank Amar Phanishayee for giving me the opportunity to put research into practice at Microsoft and collaborators at Facebook AI for giving me the opportunity to translate research into mature and production-scale engineering contributions.

I am grateful for the company of wonderful colleagues at UW-Madison: Arjun Singhvi, Arjun Balasubramanian, Anubhavnidhi Abhashkumar, Surya Teja Chavali, Kausik Subramanian, Yanfang Le, Raajay Vishwanathan, Junaid Khalid, Robert Grandl, Brent Stephens, Varun Chandrasekaran, Ayon Sen, Yuvraj Patel, Anthony Rebello, Aishwarya Ganesan, Ramnatthan Alagappan, Varsha Pendyala, Srivatsan Ramesh, Aoran Wu, ChonLam Lao. I would like to thank them for there help during research projects, for the camarederie, as well as making paper deadlines and late night work sessions less stressful.

I am grateful for my friends: Nitish Mathur, Kaushik Chandrasekaran, Vivek Saraswat, Aashrith Saraswathibhatla, Sachin Muley, Kaivalya Molugu, Raunak Bardia, Amrita Roy Chowdhury, Shaleen Deep, Swapnil Haria, Sukriti Singh, Yash Govind, Neha Govind, Samartha Patel, Akhil Guliani, Mrugank Bhat, Kunal Bhagat, Ashley Tucewicz. I thank them

for their patient support and making my stay in Madison as well as Boston a wonderful and cherishable experience.

I am grateful to all my collaborators and teachers during my pre-PhD days at IBM Research, my undergraduation at IIT Delhi, schools in Pune and Aurangabad. There guidance set the platform and made me capable enough to pursue a PhD.

Lastly, I am eternally grateful to my family. My sister, Sameedha, and my parents, Madhuri and Sharad, and others in the family have always supported me through thick and thin. They are the source of my calm self-belief and happiness. I dedicate this PhD to them as well as mentors who have tranformed my life over the years.

Abstract

Software systems that process data for meaningful insights are increasingly important in a data-driven world. The ecosystem of such data analytics systems is rich and driven by several use-cases such as relational query processing and artificial intelligence training. Each such use-case is enabled by a stack of software systems, where each system at a particular layer in the stack takes on a particular concern. Execution Planning and Scheduling are the two main concerns that are common to all data analytics systems. Execution Planning breaks the large data analytics job into smaller tasks to be executed on the cluster: a distributed network of machines. Scheduling orchestrates the execution of tasks from within a job as well as from across jobs onto the cluster. Managing these two concerns is crucial for striking a satisfactory tradeoff between fairness i.e., guaranteeing equal performance to all jobs and efficiency i.e., maximizing utilization of cluster resources. Over the years, these concerns have ossified into two different layers, separated by a fixed and static interface, governed by a separate set of developers as part of different software projects. This ossification limits the ability of the execution planner and scheduler to interact and exploit the rich optionality in decision-making available at each layer of the stack. This limits the ability to be optimally fair and efficient. In this thesis, we reimagine the interface between the layers and propose scheduler-driven strategies to co-optimize scheduling and execution planning in these three systems: (1) `QOOP`: a fair and efficient query processing system that proposes a dynamic query execution planner that interfaces with a simple fair scheduler to replan the query execution plan every time the scheduler sends a resource share update to the execution planner, (2) `Themis`: a fair and efficient GPU cluster scheduler that dynamically changes the placement of GPUs assigned to an ML training job with time, and (3) `Syndicate`: an efficient communication optimizer that improves ML training job iteration time by fragmenting communication collectives as part of execution planning and changing the scheduling order of fragmented collectives.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Software systems that process data for meaningful insights are becoming increasingly important in a data-driven world. The ecosystem of such data analyics systems is rich and driven by several unique use-cases. Figure 1.1 shows a layered view of the space of all the data processing use-cases –

- each dataset type (such as relational, images, graph, text, etc.),
- each analysis technique (such as machine learning, query processing, graph processing etc.),
- each usage mode (such as streaming, batch, etc.),
- each usage guarantee (such as fairness, cluster efficiency, etc.),
- each usage scale (such as datacenter scale, rack scale, etc.),
- each usage scenario (such as single-user, multi-tenant, etc.),

and different combinations thereof lead to several unique use-cases.

Each such unique use-case has a stack of software systems that can be composed together into a data analytics system to realize that particular use-case. These data analytics systems are more or less similar in how they compose a layered set of concerns. Figure 1.1 shows the layering of these concerns and shows the pool of software systems available at different layers in this software stack. These concerns, each of which executes a unique role, can be categorized into five different layers –

- *Datastore:* Datastores enable storage of datasets and different software system options at this layer are suitable for different use-cases dealing with different datasets.
- *Execution Framework:* Frameworks enable expression of different analysis techniques over datasets using a domain-specific language (DSL) or compute constructs (SQL, MapReduce, tensor algebra).

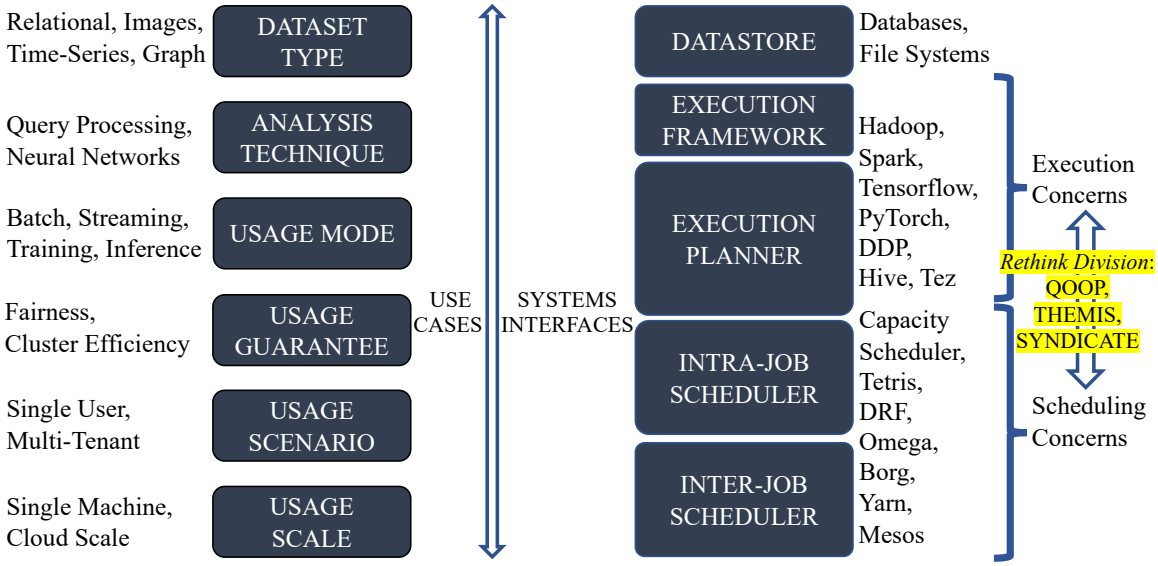


Fig. 1.1 Overview of Use-Cases, Systems and Interfaces

– *Execution Planner:* The execution planner then takes a program or a job written using this DSL and converts it to a bag of tasks (typically, as a dataflow or a directed acyclic graph (DAG) of tasks).
– *Intra-Job Scheduler:* The intra-job scheduler requests a pool of cluster resources from the cluster scheduler and enables “efficient" scheduling of these tasks onto those resources.
– *Inter-Job Scheduler:* The inter-job scheduler (also referred to as the cluster scheduler or the cluster manager) is responsible for virtualizing and allocating cluster resources to different jobs from different workloads, usually in a “fair” manner.

These layers have evolved organically over the years to support evolving use-cases. Advances in systems at different layers have been driven by different stakeholders: individual open-source contributors, research and developer communities, as well as closed-source enterprise projects. The interfaces between these different layers have also evolved organically and been standardized into fixed and static APIs by several consortiums comprising of stakeholders from systems projects from across different layers.

Efficiency and Fairness are Optimized Independently: The goal of data analytics system is to maximize efficiency under constraints of fairness. Efficiency is a measure of maximizing utilization of cluster resources. Fairness is a measure of guaranteeing equal performance to all jobs. Traditionally, efficiency is optimized by the joint actions of the execution planner and the intra-job scheduler, while fairness is optimized by the inter-job scheduler. With such a division, efficiency and fairness are optimized independently. The execution planner is unaware of the actions of the inter-job scheduler and tries to break the job into tasks in a best-effort manner. For example, for the case of batch query analytics the execution

planner (such as Volcano [6]) chooses a plan with minimum cost. The minimum cost plan typically corresponds to a plan with least execution time, however this cost fails to account for variabilities in execution time due to the actions of the inter-job scheduler and the presence of other jobs. The inter-job scheduler on the other hand takes tasks submitted by the execution planner of each job and multiplexes these tasks onto the cluster resources in a fair manner.

1.1 Joint Optimization Strategies

In this thesis, we make a case for joint optimization of efficiency and fairness. During execution planning, there are a lot of different options in which execution plan to choose. During scheduling, there are a lot of different options in what resources to allocate to each job and how many resources to allocate to each job over time. This optionality can be harnessed to enable joint optimization of efficiency and fairness and further optimize it. There are three main strategies that we explore in this work.

Top-heavy Strategy exploiting Execution Planning Optionality: The optionality in execution planning can be used to further optimize efficiency (without affecting fairness) by choosing a different execution plan by being more aware of the resources allocated to the job over time. This strategy changes the execution plans over time keeping the resource allocation across jobs fixed.

Bottom-heavy Strategy exploiting Scheduling Optionality: The optionality in scheduling can be used to change how much and how many resources are allocated to each job to further optimize efficiency (without affecting fairness) by being more aware of the resource usage characteristics of the execution plan choice made by the execution planner of different jobs. This strategy changes the resource allocation across jobs over time keeping the execution plans for all the jobs fixed.

Monolithic Strategy exploiting Cross-Stack Optionality: This strategy changes both the execution plan as well as the resource allocation over time to jointly optimize for efficiency as well as fairness by colocating scheduling and execution planning concerns in a monolithic procedure.

1.2 Thesis Contributions

In this thesis, I explore these three strategies to make data analytics systems more efficient and fair as part of three systems: `QOOP`, `Themis`, and `Syndicate`.

`QOOP` employs a top-heavy strategy for query processing workloads. Top-heavy strategy is better as each query processing job has a rich diversity in query execution plan options.

`Themis` employs a bottom-heavy strategy for ML training workloads. Bottom-heavy strategy is better as each ML training job has rich diversity in resource usage characteristics as the placement of resources allocated to each job changes.

`Syndicate` employs a monolithic strategy for ML training jobs as there is rich optionality in how different collectives from a job are executed on different communication channels as well as rich optionality in how to order collectives from a job in time.

To instrument these different strategies, I reimagine the existing division of execution and scheduling concerns and rethink the interfaces across the different layers in the stack as shown in Figure 1.1.

I now highlight the contributions of each of these systems.

1.2.1 Dynamic Query Re-Planning Using QOOP

In Chapter 2, we propose QOOP. Modern data processing clusters are highly dynamic – both in terms of the number of concurrently running jobs and their resource usage. To improve job performance, recent works have focused on optimizing the cluster scheduler and the jobs' query planner with a focus on picking the right query execution plan (QEP) – represented as a directed acyclic graph – for a job in a resource-aware manner, and scheduling jobs in a QEP-aware manner. However, because *existing solutions use a fixed QEP throughout the entire execution*, the inability to adapt a QEP in reaction to resource changes often leads to large performance inefficiencies.

This work argues for *dynamic query re-planning*, wherein we re-evaluate and re-plan a job's QEP during its execution. We show that designing for re-planning requires fundamental changes to the interfaces between key layers of data analytics stacks today, i.e., the query planner, the execution engine, and the cluster scheduler. Instead of pushing more complexity into the scheduler or the query planner, we argue for a redistribution of responsibilities between the three components to simplify their designs. Under this redesign, we analytically show that a greedy algorithm for re-planning and execution alongside a simple max-min fair scheduler can offer provably competitive behavior even under adversarial resource changes. We prototype our algorithms atop Apache Hive and Tez. Via extensive experiments, we show that our design can offer a median performance improvement of $1.47\times$ compared to state-of-the-art alternatives.

1.2.2 THEMIS: Fair and Efficient GPU Cluster Scheduling

In Chapter 3, we propose THEMIS. Modern distributed machine learning (ML) training workloads benefit significantly from leveraging GPUs. However, significant contention

ensues when multiple such workloads are run atop a shared cluster of GPUs. A key question is how to fairly apportion GPUs across workloads. We find that established cluster scheduling disciplines are a poor fit because of ML workloads' unique attributes: ML jobs have long-running tasks that need to be gang-scheduled, and their performance is sensitive to tasks' relative placement.

We propose THEMIS, a new scheduling framework for ML training workloads. It's GPU allocation policy enforces that ML workloads complete in *a finish-time fair* manner, a new notion we introduce. To capture placement sensitivity and ensure efficiency, THEMIS uses a two-level scheduling architecture where ML workloads bid on available resources that are offered in an *auction* run by a central arbiter. Our auction design allocates GPUs to winning bids by trading off fairness for efficiency in the short term, but ensuring finish-time fairness in the long term. Our evaluation on a production trace shows that THEMIS can improve fairness by more than 2.25*X* and is ~5% to 250% more cluster efficient in comparison to state-of-the-art schedulers.

1.2.3 Better Together: Jointly Optimizing ML Collective Scheduling and Execution Planning using SYNDICATE

In Chapter 4 we propose SYNDICATE. Emerging ML training deployments, such as Deep Learning Recommendation Model (DLRM) training, are trending towards larger models, and hybrid-parallel training that is not just dominated by compute-intensive all-reduce for gradient aggregation but also bandwidth-intensive collectives (e.g., all-to-all). These emerging collectives exacerbate the communication bottlenecks despite heterogeneous network interconnects with ample multipath opportunities. In this work, we propose SYNDICATE, a systematic, general framework to minimize communication bottlenecks and speed up training for both state-of-the-art and future large-scale models and interconnects. SYNDICATE proposes a novel abstraction, the motif, to break large communication work as smaller pieces as part of execution planning. SYNDICATE also does joint optimization of scheduling and execution planning by rethinking the interfaces in the networking systems stacks used for ML training. Motifs afford greater flexibility during scheduling and the joint optimizer exploits this flexibility by packing and ordering communication work so as to maximize both network utilization and overlap with compute. This improves performance of DLRM training by 38–74%.

Chapter 2

Dynamic Query Re-Planning Using QOOP

2.1 Introduction

Batch analytics is widely used today to drive business intelligence and operations at organizations of various sizes. Such analytics is driven by systems such as Hive [6] and SparkSQL [27] that offer SQL-like interfaces running atop cluster computing frameworks such as Hadoop [5] and Spark [122]. Figure 2.1 shows the key layers of data analytics stacks today. At the core of these systems are query planners (QPs) , such as Calcite for Hive [4] and Catalyst for SparkSQL [27]. QPs leverage data statistics to evaluate several potential query execution plans (QEPs) for each query to determine an optimized QEP. The optimized QEP is a DAG of interconnected stages, where each stage has many tasks. An execution engine then handles the scheduling of these tasks on the underlying cluster by requesting resources from a scheduler. The scheduler allocates resources considering a variety of metrics such as packing, fairness, and job performance [54, 55, 124].

To improve query performance, existing works have primarily looked at optimizations limited to specific layers in the data analytics stack. Some of them [121, 124, 25, 53, 55, 54, 95] have focused on improved scheduling given the optimized QEP by incorporating rich information, such as task resource requirements, expected task run times, and dependencies. Others have considered improving the QP to take into account resource availability at query launch time (in addition to data statistics) to find good resource-aware QEPs [111, 112].

We argue that these state-of-the-art techniques fall short in *dynamic environments*, where resource availability can vary significantly over the duration of a job's lifetime. This is because existing techniques are *early-binding* in nature – a QEP is pre-chosen at query

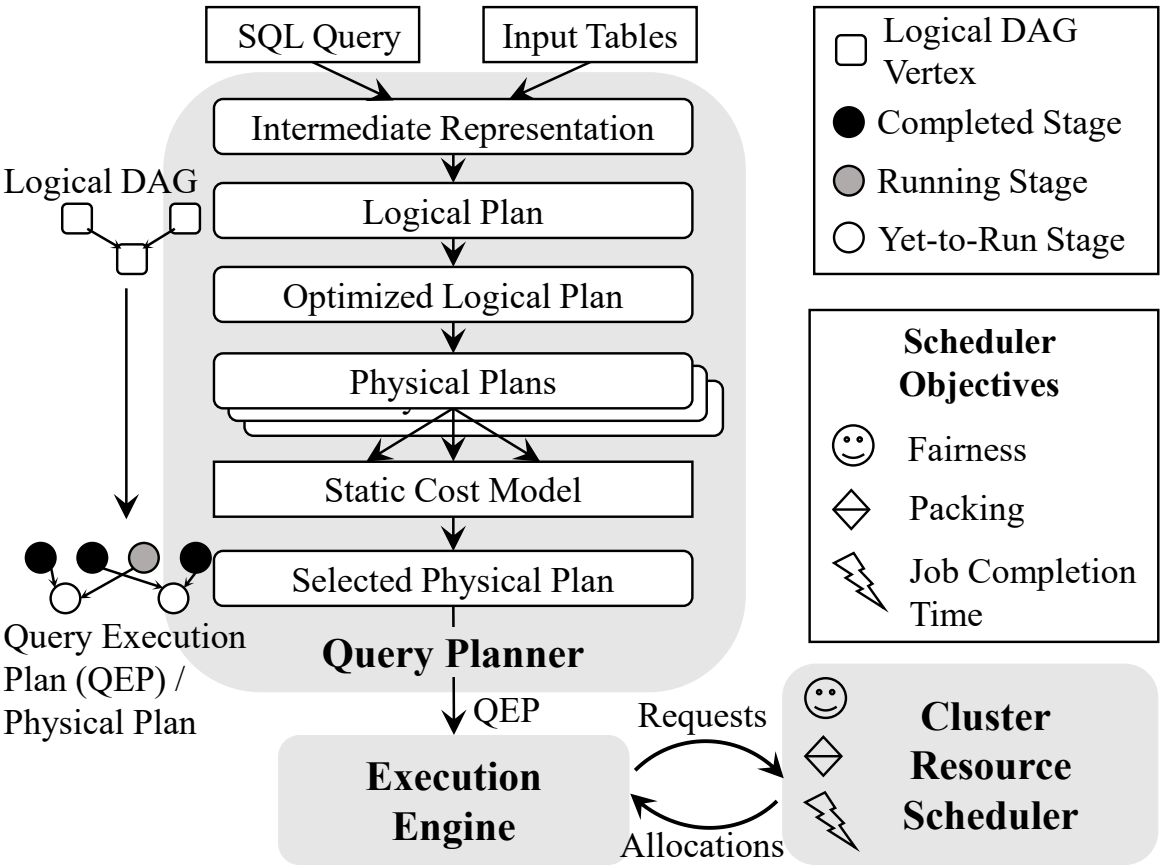


Fig. 2.1 Traditional batch query execution pipeline.

launch time and the QEP's low-level details (e.g., the physical tasks, task resource needs, dependencies) are used to make scheduling decisions (which tasks to run and when). This fundamentally leaves limited options to adapt to resource dynamics. Our work makes a case for *constant query replanning* in the face of dynamics. Here, a given job switches query plans during its execution to adapt to changing resource availability and ensure fast completion.

Dynamic resource variabilities can arise in at least two situations: (i) running multiple jobs on small private clusters, which is a very common use-case in practice [7]; and (ii) leveraging spot market instances for running analytics jobs, which is an attractive option due to the cost savings it can offer [78, 102, 127, 64]. We empirically study resource changes in these situations in Section 4.3.

To enable effective adaptation in these situations, we develop and analyze *strategies for query replanning*. We prove two basic results: (1) When dynamically switching QEPs, it is important for a query to potentially *backtrack* and forgo already completed work. Given imperfect knowledge of future resource availability, a query's performance can be arbitrarily bad without backtracking. (2) A *greedy algorithm* – which always picks a QEP offering the best completion time assuming current allocation persists into the future – performs well. We

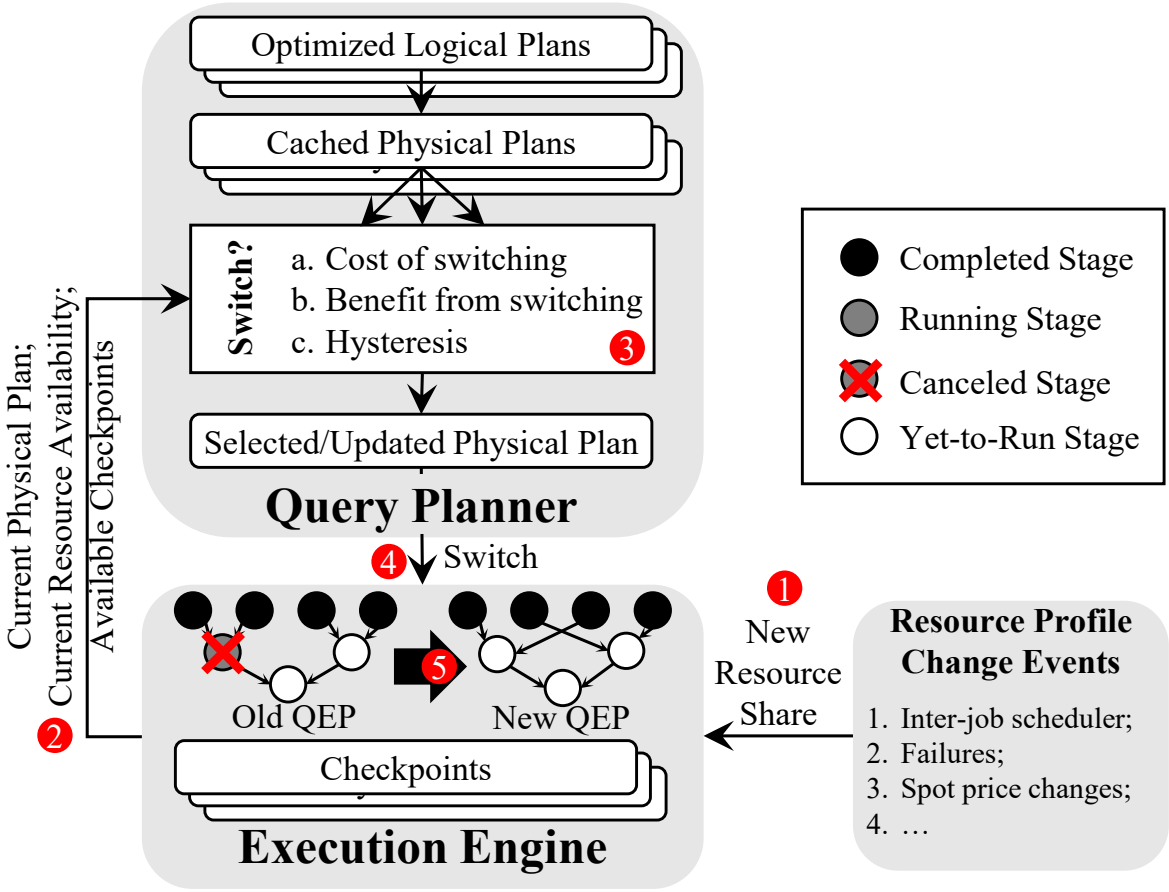


Fig. 2.2 Dynamic replanning in action using QOOP. Omitted part of the query planner is similar to that of Figure 2.1.

prove that the greedy algorithm has a competitive ratio of 4; the lower bound for any online algorithm is 2.

To realize the aforementioned replanning strategies in practice, we eschew the early binding in today's approaches. Instead, we propose a new system, QOOP, that has the following radically different division of labor and interfaces among the layers of analytics stacks (Figure 2.2):

- The cluster scheduler implements *simple* cluster-wide weighted resource shares and *explicitly* informs a job's execution engine of changes to its cluster share. The cluster share of a job is defined as the total amount of each resource divided by the number of active jobs. During a jobs's execution, our scheduler tracks a job's current resource usage – measured as the maximum of the fractions of any resource it is using – and allocates freed up resources to the job with the least current usage, emulating simple max-min fair sharing. Thus, the scheduler decouples the feedback about cluster contention – this helps queries replan and adapt – from task-level resource allocation, which is instantaneously max-min fair.

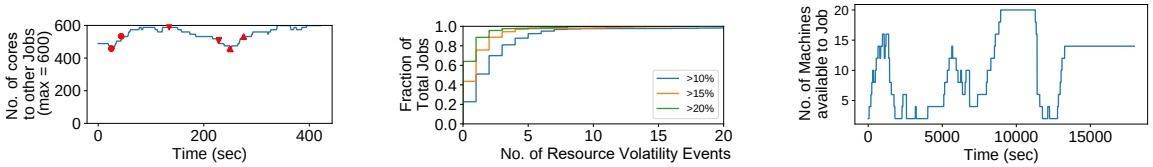
• When resource shares change *significantly*, the query planner compares a query's remaining time to completion based on its current progress against its expected completion time from replanning and switching to a different plan. It uses a model of task executions and available checkpoints in the execution engine to make this decision. It picks a better QEP to switch to (if one exists), and informs the execution engine of the new set of tasks to execute and existing ones to revoke.

• The *execution engine* supports the query planner by informing it of the query's current progress and maintaining checkpoints of the query's execution from which alternate QEPs' computation can begin.

Overall, QOOP pushes complexity up the stack, out of cluster schedulers – where most of the scheduling complexity exists today – and into a tight replanning feedback loop between the query planner and the execution engine. We show that the resulting *late binding* enables better dynamic query adaptation.

We prototype QOOP by refactoring the interfaces between Hive, Tez, and YARN. Our evaluations on a 20-node cluster using TPC-DS queries show that QOOP's dynamic query replanning and simple scheduler outperform existing state-of-the-art static approaches. From a single job's perspective, QOOP *strictly* outperforms a resource-aware but static QP. For example, when resource profiles fluctuate rapidly, with high volatility, QOOP offers more than 50% of the jobs improvements of 1.47× or more; 10% of the jobs see more than 4× gains! We also use QOOP to manage the execution of multiple jobs on a small 20-node private cluster. We find that QOOP performs well on all three key metrics, i.e., job completion times, fairness, and efficiency, by approaching close to the individual best solutions for each metric.

2.2 Background and Motivation



(a) Jobs' view of cluster resources

(b) Distribution of resource volatility

(c) Number of machines reserved by a job in a spot market

Fig. 2.3 Analysis of resource perturbations in a shared cluster and spot market. The gaps between each pair of the same symbols in (a) demarcate one resource volatility event.

In this section, we highlight multiple sources of resource dynamics in a cluster (§2.2.1), discuss the opportunities lost from not being able to switch a query's plan in response to resource dynamics (§4.2), and why the existing interfaces between cluster schedulers, execution engines, and query planners make dynamic switching difficult (§2.2.3).

2.2.1 Resource Dynamics in Big Data Environments

Modern big data queries run in dynamic environments that range from dedicated resources in private clusters [40, 32] and public clouds [2] to best-effort resources put together from spot markets in the cloud [78, 102, 127, 64].

In case of the former, resources are arbitrated between queries by an inter-job scheduler [54, 48, 55, 124, 25]. As new jobs arrive and already-running jobs complete, resource shares of each job are dynamically adjusted by the scheduler based on criteria such as fairness, priority, and time-to-completion. Although in large clusters, such as those run by Google [40, 46] and Microsoft [32], individual job arrivals or departures have negligible impact on other jobs, most on-premise and cloud-hosted clusters comprise less than 100 machines [7, 10] and run only a handful of jobs concurrently. A 2016 Mesosphere survey [7] found that 96% of new users, and 75% of regular users use fewer than 100 nodes. A single job's arrival or completion in such scenarios can create large resource perturbations.

To better highlight resource perturbations in small clusters, we ran a representative workload on a 20-node cluster managed by Apache YARN. The cluster uses the Tetris [53] cluster scheduler, and it can concurrently run 600 containers at it's maximum capacity (1 core per container in a 600 core cluster). For our workload, we use the TPC-DS [18] workload, where jobs arrive following a Poisson process with an average inter-arrival time of 20 seconds. The average completion time per job is around 500s. We pick a job executed in the cluster and show it's view of cluster resources in Figure 2.3a. Specifically, we show the number of cores allocated (out of a maximum of 600) to all the *other* jobs running concurrently. During its lifetime, the job we picked experiences resource volatility – we call an x% increase or decrease in resource (number of cores in this case) over some period of time as an x% resource volatility. In Figure 2.3a, we identify 15% resource volatility within uniquely shaped red markers; e.g., the region between two solid red circles indicates one such 15% resource volatility. The job observes 3 such resource volatility events during its lifetime (identified within similarly shaped markers). To understand resource volatility as observed by different jobs for different resource volatility magnitudes (different values of x), in Figure 2.3b, we plot a CDF of the number of resource volatility events seen by each of the individual jobs in our workload for three values of x = 10%, 15% and 20%. We observe that almost 78% of the

jobs experience at least one 10% resource volatility event during their lifetime, and 20% of the jobs see at least 4 resource volatility events of 10% or more.

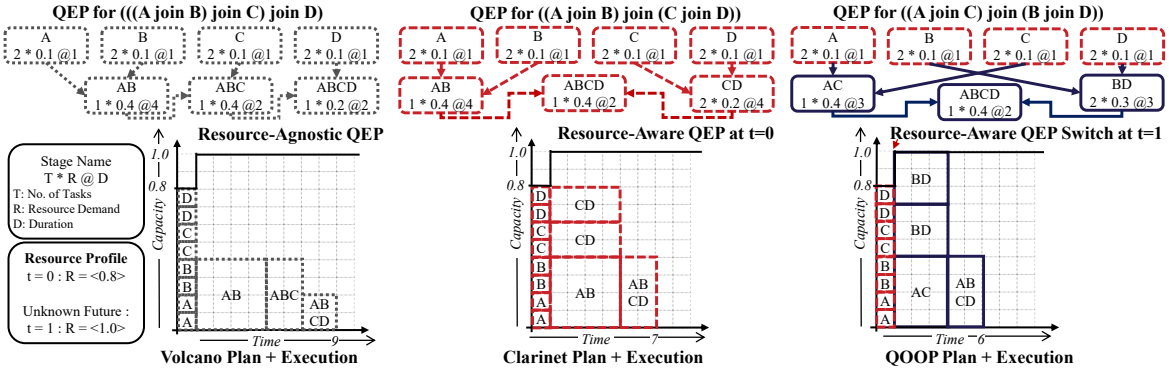


Fig. 2.4 Comparison of query planners. The QEPs shown correspond to three different logical plans for the query - $A \bowtie B \bowtie C \bowtie D$. Volcano chooses the QEP on the left (as this plan has the least resource consumption) that completes at $t = 9$. Clarinet chooses an optimal resource-aware QEP (under static resources) at $t = 0$ that completes at $t = 7$; however, it ceases to be resource-aware when available resources change at $t = 1$. QOOP re-plans to switch to a new plan at $t = 1$ and completes the fastest at $t = 6$.

At the other extreme, running jobs on spot instances – with their input on blob storage like Amazon S3 [3] – is becoming common because spot instances offer an attractive price point [78, 102, 127, 64]. However, cloud providers can arbitrarily revoke spot instances, which can cause perturbations in the number of machines available to a job. We now empirically examine the extent of such potential resource variations as experienced by a resource-intensive, batch job that runs for five hours. We use the spot-market price trend for `i3.2xlarge` instance type in Amazon EC2 cluster in the `us-west-2c` region for the time period from 17:00 UTC to 21:00 UTC for September 21, 2017. We also assume that the job has a budget of 5$/hour and that spot instances that were reserved at less than the current spot market price are taken away immediately. The spot instance prices typically update every minute. We use a simple cost-saving bidding strategy where the job progressively adds 2 spot instances every minute, provided the budget is not exceeded, by bidding at a price 5% over the current spot market price. Under such a bidding strategy and a budget of 5$, the maximum number of machines that the job gets is 20 and the minimum is 2. The number of spot instances available to the job over time is shown in Figure 2.3c. We make the following observations. First, the job experiences many perturbations in the number of machines, which is especially true with cost-saving bidding strategies. Second, the magnitude of perturbation is the largest around the 3 hour mark when the spot market instance price reaches a maxima of 0.5828$/hour and all but 2 machines are revoked. Finally, throughout the entire duration

of the job, the job experiences 60, 53 and 40 resource volatility events of 10%, 15% and 20% respectively.

Other common sources of resource fluctuations include machine/rack failures, planned or unplanned upgrades, network partitions, etc. [30, 115].

2.2.2 Query Execution Today: Fixed Plans

Regardless of the extent of resource dynamics, existing approaches *keep the query plan fixed throughout the entire duration of a query's execution*. However, these approaches do vary in terms of what information they use during query planning and how they execute a query.

Resource-Agnostic Query Execution: A large number of today's data-analytics jobs are submitted as SQL queries via higher-level interfaces such as Hive [6] or Spark SQL [27] to cluster execution engines (Figure 2.1).

A cost-based optimizer (CBO) examines multiple equivalent logical plans for executing a query, and leverages heuristics to select a good plan, also called a query execution plan (QEP).[1] The QEP represents the selected logical plan and its relational operators as a *job* with a directed acyclic graph (DAG) of computation stages and corresponding tasks that will be executed by the underlying execution engine on a cluster of machines. Given the chosen QEP – also called the physical plan – the execution engine interacts with the cluster resource scheduler in a repeated sequence of resource requests and corresponding allocations until all the tasks in the physical plan of the job complete. *Crucially, the optimizer's heuristics are based on data statistics and not resource availability; thus, it is resource-agnostic.* An example is the Volcano query planner in Hive [6]. Figure 2.4 shows a Volcano-generated plan – a QEP corresponding to a "left deep" plan – that is preferred by the Volcano CBO based on data statistics.

Resource-Aware QEP Selection: Given the obvious inflexibility of resource-agnostic query optimization, some recent works [111, 112] have proposed resource-aware QEP selection. In this case, the CBO takes available resources into account before selecting a QEP and handing it over to the execution engine. While this is an improvement over the state-of-the-art, the execution engine still runs a fixed QEP even when resource availability changes over time. An example of a resource-aware planner is Clarinet [111]. As shown in Figure 2.4, the Clarinet plan is chosen based on the resources available at $t = 0$. When the resources change at $t = 1$, the static plan ceases to be the best.

Room for Improvement: Instead of sticking to the original resource-aware or -agnostic QEP throughout execution, one can find room for improvements by switching to a new QEP

[1] Some optimizers consider a narrow set of resources, such as the buffer cache or memory, but ignore disk and network [6].

on the fly based on resource changes. For example, when the available resource increases at $t = 1$ in Figure 2.4, we can switch to a different join order – $(A \bowtie C) \bowtie (B \bowtie D)$ instead of $(A \bowtie B) \bowtie (C \bowtie D)$ – and further decrease query completion time.

Although this is a toy example, overall benefits of dynamic query re-planning improve with the complexity of query plans, magnitudes of resource volatility, and pathological fluctuations of resources due to unforeseen changes in the future (§4.6).

2.2.3 Scheduler Constraints on QEP Switching

Unfortunately, today's cluster schedulers and their interfaces with the execution engine and the query planner make resource-aware QEP switching challenging.

On the one hand, *existing schedulers provide little feedback to jobs about the level of resource contention in a cluster* – today, jobs simply ask the scheduler for resources for runnable tasks and the scheduler grants a subset of those requests. Consequently, it is difficult for a job to know how to adapt in an informed manner to changing cluster contention or resource availability. One may think that jobs can infer contention by looking at the rate at which their resource requests are satisfied. However, such an inference mechanism can be biased by the resource requirements of the tasks in the currently chosen QEP instead of being correlated to the level of contention.

On the other hand, *scheduling decisions are tied to the intrinsic knowledge of job physical plan*. Schedulers are tasked with improving inter-job and cluster-wide metrics, such as fairness, makespan, and average completion time [48, 53, 45, 54, 55]. For example, DRF tracks dominant resources, which relies on the multi-dimensional resource requirements of physical tasks. Others [55, 53, 54] go further and combine resource requirements with the number of outstanding tasks and dependencies to estimate finish times using which scheduling decisions are made. The tight coupling of schedulers with pre-chosen QEPs constrains the scheduler to make decisions to match resources with the demands imposed by the pre-chosen QEP's tasks.

Overall, neither the job nor the scheduler has any way of knowing whether picking a different QEP with a very different structure and task-level resource requirements would have performed better – w.r.t. per-job or cluster-wide metrics – under resource dynamics.

2.3 QOOP Design

In this work, we argue for breaking the constraints of fixed QEPs, and we make a case for continuous query re-planning by rethinking the division of labor between cluster schedulers,

execution engines, and query planners. We first give an overview of our design (§2.3.1) and then present its three key components: a simple max-min fair scheduler (§2.3.2), an execution engine design to track additional states needed to speed up dynamic re-planning (§2.3.3), and a greedy QP that performs well with provable performance guarantees (§2.3.4).

2.3.1 Design Overview

The state-of-the-art approaches for improving query performance universally argue for pushing more complexity into the inter- and intra-job scheduling to achieve efficiency and improve job performance; by design, this prevents adaptation at the query level. Instead, to achieve replanning, we propose a significant refactoring. (1) We advocate having a simple max-min fair scheduler that effectively does "1-over-n" allocation of every resource across n jobs. (2) Jobs are informed as soon as their share changes due to changing n or machine/rack failures. (3) We push re-planning complexity *up the stack*, maintaining a *dynamic re-planning feedback loop* between the query planner and the execution engine: based on changes to the share, the planner – with help from the execution engine – determines if a better QEP exists and how to switch to it.

We choose this work division because each instance of an application framework today implements its own query planner and execution engine (e.g., both implemented in the Job Manager in case of frameworks using Apache YARN), whereas *all* jobs running in a cluster share the same centralized resource scheduler (i.e., the Resource Manager in Apache YARN). Our division of labor has the benefit of enabling many different applications with their intrinsic continuous re-planners to effectively run atop our simple cluster scheduler. For simplicity, our work focuses just on re-planning batch SQL queries.

Figure 2.2 presents our architecture with the sequence of actions that take place on a resource change event: ❶ The cluster scheduler or the resource manager notifies the execution engine of its new resource share (§2.3.2). ❷ The execution engine, in turn, notifies the query planner of the current state, which includes the current QEP it is executing along with its progress, current resource availability it received from the scheduler, and the available set of checkpoints it is maintaining (§2.3.3). ❸ Given this information, the query planner must determine whether switching to a new plan is feasible (considering available checkpoints, cost of possible backtracking, and hysteresis) (§2.3.4). ❹ If the decision is yes, then it informs the execution engine of the new QEP. ❺ Finally, the execution engine will switch to the new QEP; if required, it will cancel some already-running stages and tasks.

Realizing dynamic re-planning raises a few key algorithmic questions. First, what is a good switching strategy when resources change? A simple and easy-to-implement choice is Greedy: i.e., always pick the QEP that offers the least estimated finish time assuming

the new resource availability persists into the future. Does this offer good properties under arbitrary resource fluctuations? Second, switching from a QEP with partial progress to a new one that needs to be started from scratch necessarily wastes work. Is this "backtracking" necessary? In Section 2.4, we show that the simple Greedy approach performs well, and that backtracking is essential.

Before presenting the analysis, in Sections 2.3.3 and 2.3.4, we discuss key systems issues that arise in supporting greedy behavior with backtracking: How to estimate the relative runtimes of different QEPs? How to preserve work to support backtracking and leverage already computed work when switching to a new QEP? We start by outlining the functionality of our inter-job scheduler next.

2.3.2 Cluster Resource Scheduler

Our inter-job scheduler is simple (Pseudocode 1). For each job, our scheduler tracks the job's current (weighted) share in every resource dimension, i.e., the total fraction of the resource that all currently running tasks of the job are using. The scheduler computes the *current share* of the job as the maximum of these fractions taken over all resources. When a resource is freed on a machine, our scheduler simply assigns it to the job with the lowest current share that can run on that machine, emulating simple instantaneous max-min fair allocation of resources across jobs, similar to [21, 48]. This is shown in lines 4–8. We are algorithmically similar to DRF, but differ in API. When resources become available DRF allocates to the job with least dominant-share; QOOP informs each job of its dominant share on resource-change events.

To enable re-planning, we introduce two changes to the interface between the scheduler and the execution engine. First, we do not require the execution engine to propagate the entire QEP to the cluster scheduler. Decoupling the QEP from the resources assigned to a job has the desirable property that the execution engine can change the QEP without affecting its fair share of resources, which is not the case for the state-of-the-art techniques [48, 55, 54].

Second, we introduce feedback from the cluster scheduler to the execution engine (line 8 in Pseudocode 1). Whenever the current cluster share of a job changes, the scheduler informs the job's execution engine. The cluster-wide fair-share informs each job of its minimum resource share given the current contention in the cluster. This acts as a *minimum resource guarantee* for the query planner when determining whether to re-plan in order to finish faster. In fact, any scheduler that can offer feedback in the form of an eventual minimum resource guarantee of resources to each job is compatible with QOOP.

Pseudocode 1 Cluster Scheduler

```
1: 𝕁                                          ▷ active jobs prioritized by lowest current share
2: R⃗                                          ▷ total cluster resource capacity
3: U⃗                                          ▷ consumed cluster resource portion

4: procedure MaxMinFairScheduler
5:     pick first J ∈ 𝕁                       ▷ triggered when R⃗ − U⃗ > 0⃗
6:     allocate demand D⃗_i ∈ J s.t. max_{i,m} D⃗_i · (R⃗_m − U⃗_m)
7:     update 𝕁

8: procedure ResourceFeedback(Event 𝔼)
9:     𝕁 = 𝕁 ⊕ getJobChanges(𝔼)
10:    R⃗ = R⃗ ⊕ getResourceChanges(𝔼)
11:    fairShare = R⃗ / |𝕁|
12:    for all J_k ∈ 𝕁 do
13:        SendResourceFeedbackUpdate(J_k, fairShare)
```

2.3.3 Execution Engine

We discuss how job execution engine redesign can enable query re-planning, specifically backtracking.

Task Execution: Given a job QEP DAG (i.e., the output of a query planner), the execution engine executes tasks by interacting with the cluster scheduler while maintaining their dependencies. To determine the order of task execution, it can simply traverse the DAG in a breadth-first manner [122, 27] or use a multi-resource packing algorithm such as Tetris [53]. In QOOP, we use Tetris.

Checkpointing for Potential Switching Points: On any multi-resource update from the cluster scheduler, the execution engine relays the updated resource vector to the query planner to evaluate the possibility of switching to a different QEP. Determining whether to actually switch to a new QEP relies on multiple factors (§2.3.4). A major one is finding the suitable point(s) in the currently executing DAG to switch from. One may consider that switching from the currently executing stage or its immediate parent stage(s) would suffice. However, we prove in Section 2.4 that *backtracking* to ancestor stage(s) is essential for competitively coping with unknown future resource changes.

Consequently, QEP switching may not just re-plan the future stages of the query, but it requires the ability to checkpoint past progress and switch to a different QEP from an ancestor stage that was executed in the past. To enable this, the execution engine needs to checkpoint past progress for all the different QEPs it has executed thus far. Each checkpoint

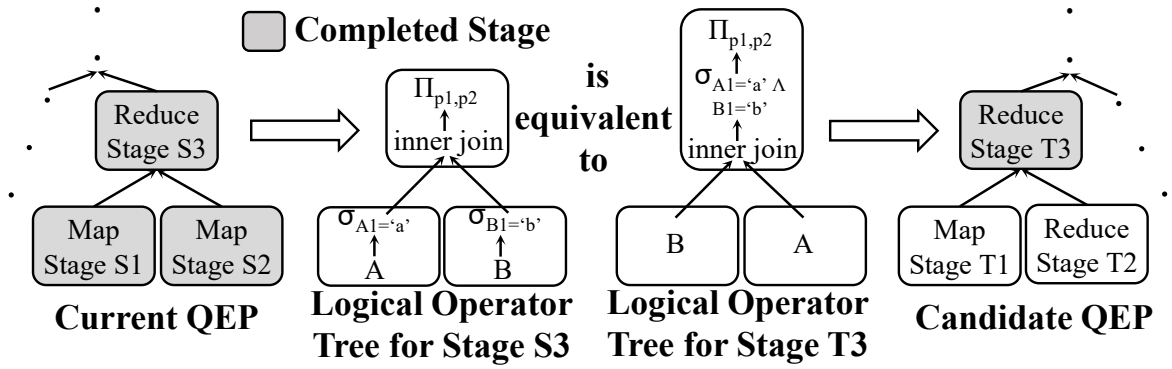


Fig. 2.5 Progress propagation. First, we obtain logical operator trees for stages $S3$ and $T3$ from provenance. Stages $S3$, $T3$ are deemed equivalent as their logical operator trees are equivalent.

includes the intermediate outputs of completed tasks. Note that checkpointing of intermediate data is common in modern execution engines – disk-based frameworks write intermediate data to disks [40, 8], whereas in-memory frameworks periodically checkpoint to avoid long recomputation chains [122, 123]. QOOP can use this existing checkpointing.

Switching the QEP: The call back to the query planner (upon resource updates) is asynchronous. While the query planner is evaluating possible alternatives, the execution engine continues on with the current plan. When the query planner suggests a change, the execution engine revokes the resource requests for runnable tasks not belonging to the new QEP. Additionally, the execution engine may abort running tasks not belonging to the new QEP. Thereafter, the execution engine resumes running tasks from the most-recent set of checkpoints for the new QEP.

2.3.4 Query Planner (QP)

In Section 2.4, we show that effective re-planning requires backtracking and that a greedy approach to re-planning results in a competitive online algorithm. Here, we present the details of how our query re-planner implements greedy re-planning by leveraging backtracking.

We introduce two key changes to the design of traditional QPs; neither requires extensive modifications. First, instead of discarding intermediate computations to explore and choose a particular QEP, we generate and cache several candidate QEPs. The cached QEPs later aid us in dynamic query re-planning. We also annotate each QEP with provenance, which consists of the original logical plan the QEP was derived from and the list of logical operators associated with each stage of the QEP. Figure 2.5 shows the provenance of each stage of a QEP.

Second, unlike traditional QPs [4, 111], our QP is made aware of the underlying resource contention to accurately predict runtimes for each QEP and greedily switch to the QEP

with minimum completion time. To do so, we extend the interface between the QP and the execution engine so that the QP receives parameters to its dynamic cost model – the current resources available to the job (the share that the execution engine obtains from the cluster scheduler), the intra-job scheduling logic (packing), the progress of the current QEP and the available set of checkpoints.

Whenever the query planner receives a notification about resource changes from the execution engine, it triggers a cost-based optimization that involves predicting the completion times of all the QEPs and greedily switching to the QEP with earliest completion time. There are two steps to evaluate a particular QEP: progress propagation and completion time estimation.

Progress Propagation: To evaluate a candidate QEP, the QP first evaluates the work in the candidate QEP that is already done by the currently running QEP. It does so by identifying common work between the tasks of the candidate QEP, the running tasks of the current QEP, and the current set of checkpoints. We refer to this as *progress propagation*, and it is crucial in evaluating which candidate QEP to switch to and where to execute it from.

To identify common work as part of progress propagation, we identify equivalence between the stages of a candidate QEP and the set of checkpointed stages and the current running stages of the current QEP. To evaluate equivalence between two stages we generate the stages' logical operator trees using the provenance associated with each QEP. Two stages are deemed equivalent if their logical operator trees are equivalent. Equivalence of logical operator trees is evaluated using standard relational algebra equivalence rules. This is illustrated in Figure 2.5.

Completion Time Estimation: Next, we perform a simulated execution of the remaining tasks in the candidate QEP being evaluated (i.e., candidate QEP tasks whose work is not captured in the currently running QEP). Using the scheduling algorithm of the intra-job scheduler, i.e., Tetris, the remaining tasks are tightly packed in space and time given the current available resources. This yields an estimate for this QEPs completion time assuming that the current resource availability will persist in the future.

After evaluating the completion times of all candidate QEPs, query planner triggers a query plan switch if it finds a QEP that finishes faster than the currently running QEP. To avoid unnecessary query plan flapping, we add *hysteresis* by having a threshold on the percentage improvement of the query completion time – a query plan switch is triggered only if improvements exceed this threshold.

In case of a switch, the query planner sends the new QEP to the execution engine. This QEP is modified from its original form so that the DAG now contains the checkpoints as

input stages, marks the running stages it shares with the running stages of the current QEP, and identifies the dataflow from these to the remaining stages.

2.4 Analysis

We now present analysis of the query planner (QP; Section 2.3.4). Each query has several alternative query execution plans (QEPs). We motivate the choices made in the query replanning algorithm regarding *why, when and which* QEP to switch to during the execution of a query in response to the resource allocations made by the scheduler to the query. This is an online algorithm since it operates without the knowledge of future resource allocations. We analyze the performance of our online algorithm in the form of its competitive ratio. Our goal is to argue that our online algorithm performs well no matter the sequence of resource allocations made to the query. We will compare our online algorithm's performance against an hindsight optimal (a.k.a. offline) algorithm which chooses the single best QEP knowing the entire sequence of resource allocations made to the query. The competitive ratio is the ratio of the performance of the online algorithm to that of the hindsight optimal (a.k.a. offline) algorithm. We provide a precise measure of comparison shortly (Section 2.4.3).

2.4.1 Notation and Assumptions

Notation: We represent each QEP as $a \times b$. This denotes a QEP with a bag of b tasks, each task needing a resource-units (e.g., number of cores) and each task completing in 1 step. The total work for this QEP is denoted by w and is equal to ab.
Assumptions: For the upper and lower bounds on performance, we assume an adversarial scheduler that can look at the algorithm's choices in the previous steps and change future resource allocation in a worst-case manner. We require that the QP has the ability to *backtrack* a QEPs execution i.e., the QP can checkpoint each completed task in a QEP and any completed task need not be re-executed when the QP decides to switch back to and resume the execution of that QEP. We also assume that backtracking does not incur any overheads; in other words that our analysis ignores system-level costs (time spent and compute/memory used) in writing checkpoints and reading from checkpoints during a QEP switch.

2.4.2 Motivating Example

We motivate *why* QEP switching and specifically backtracking is necessary to obtain a bound on the performance of our online algorithm.

Our toy examples, with large work-differences in QEPs, serve to show that if the online algorithm does not make good decisions then its performance can become unboundedly worse.

Example 2.1.
QEP switching is necessary. *Consider a query with two QEP choices: the first one being* 2×2 *and the second one being* 1×100*. Suppose that the scheduler starts by giving the query* 2 *resource-units in the first step. We also suppose that the query cannot switch QEPs.*
CASE-1: *If the query starts running the* 1×100 *QEP, the scheduler gives it another* 2 *resource-units in the second step. With this allocation, the optimal choice would be to run the* 2×2 *QEP, finishing in two steps and performing only* 4 *units of work. The online algorithm instead performs* 100 *units of work if it continues to use the* 1×100 *QEP.*
CASE-2*: On the other hand, if the query starts running the* 2×2 *QEP, the scheduler switches to a resource allocation of* 1 *resource-unit second step and onwards. Now the* 2×2 *QEP is stalled. Unless the algorithm switches to the* 1×100 *QEP, it is unable to finish.*
QEP backtracking is necessary. *Backtracking helps* avoid stalling, ensures fast completion, and bounds wasted work. *We continue the previous example. As before, the scheduler continues to be adversarial. It allocates* 1 *and* 2 *resource-units in the next step whenever the query is executing* 2×2 *and* 1×100 *QEP in the current step, respectively. Also, we now suppose that the query has the ability to switch QEPs but not backtrack i.e., no ability to checkpoint and resume QEPs from checkpoint.*
We continue from where we left-off in the previous example i.e., CASE-2 *where the query is executing the* 2×2 *QEP and the scheduler allocates* 1 *resource-unit in the second step. With the ability to switch, to avoid stalling, the query switches to the* 1×100 *QEP in the second step. Without backtracking, the query has to restart execution of* 1×100 *QEP from the beginning. Now on switching to the* 1×100 *QEP, the adversarial scheduler gives the query* 2 *resource-units third step onwards. This leads us back to* CASE-1*. If the QEP continues with the* 1×100 *QEP it leads to* slower completion*.*
If instead the query switches back to 2×2 *QEP in the third step, without backtracking the QEP restarts execution from the beginning and the adversarial scheduler gives the query* 1 *resource-unit fourth step and onwards. This is* CASE-2 *all over again. We can now see that, without backtracking, the query flips between* CASE-1 *and* CASE-2 *and* stalls infinitely with unbounded wasted work*. Even if the query decides to limit wasted work by stopping the switch to* 2×2 *QEP, complete execution of* 1×100 *QEP to completion leads to* 100 *units of additional work and* 100 *additional steps. This leads to* slower completion *as in* CASE-1*.*
If the query could backtrack – we would have only one additional task to run from the 2×2

QEP in the third step and the query would complete execution in the third step with just 1 *units of wasted work.*

2.4.3 Competitive Ratio

A natural way to compare the performance of our algorithm against the hindsight optimal algorithm is to compare the time each algorithm takes to complete the query. As the next example shows, this is not a meaningful comparison, because the scheduler has the power to starve the online algorithm after a single bad choice.

Example 2.2. Starvation. *Consider the above example again. As before, the scheduler starts by giving the query* 2 *resource-units in the first step. If the query starts running the* 1×100 *QEP, the scheduler gives it another* 2 *resource-units in the second step, and then gives no more resources to this query in subsequent steps. Regardless of whether the query continues running the* 1×100 *QEP or switches to the* 2×2 *QEP in the second step, the query is unable to finish the work and stalls. Its completion time is unbounded. With the same allocation of resources, the hindsight optimal algorithm could have finished the query by just running the* 2×2 *QEP.*

On the other hand, say the query starts running the 2×2 *QEP and the scheduler gives* 1 *unit resource for the next* 99 *steps and then gives no more resources. Once again no matter what the online algorithm does, it cannot complete the query. However, the hindsight optimal algorithm would have been able to complete the query given these resources.*

In each of the cases in the above example, the scheduler could stall the query for an unlimited time, whereas the hindsight optimal algorithm terminates in bounded time. In order to allow for some wasted work due to the online nature of the algorithm, the scheduler must provide more resources to the online algorithm than just the minimum necessary for the hindsight optimal algorithm. To formalize this, we will compare the completion time of the online algorithm to that of an hindsight optimal algorithm that is required to perform extra work.

Definition 2.1. Competitive Ratio. *We say that an online QEP selection algorithm achieves a competitive ratio of* α *if for any query and any sequence of resource allocations, the completion time achieved by the online algorithm is at most equal to the completion time of an offline optimum that runs* α *back-to-back copies of the query.*

We note that α above does not have to be an integer.

Pseudocode 2 Online Query Planning Algorithm

Input n QEPs, $a_i \times b_i$, with $a_1 < a_2 < a_3 < \cdots < a_n$

1: Let $w_i = a_i b_i$ denote the total work of QEP i.
2: **for all** $i \in [n]$ **do**
3: **if** $w_i > \frac{1}{2} w_{i-1}$ **then** remove QEP i from the list.
4: At every step, given the current resource allocation a, consider all QEPs with $a_i \leq a$. Of these, run the QEP with the least remaining processing time, breaking ties in favor of the QEP with the smallest a_i.

2.4.4 Bounds for the Competitive Ratio

We show that no online algorithm can achieve a competitive ratio < 2. Proofs for the theorems below can be found in Appendix A.1.

Theorem 2.1. *No online query planning algorithm can achieve a competitive ratio of* $2 - \varepsilon$ *for any constant* $\varepsilon > 0$ *when the resource allocation is adversarial.*

Our query planning algorithm corresponding to the simplifying assumptions in Section 2.4.1 is formally described above. It is greedy and at every step runs the QEP with the least remaining completion time with the assumption that the resource allocation persists forever. Also, it is "lazy" as it switches QEPs only when the resource allocation changes. Our overall approach in Sections 2.3.4 and 2.3.3 is a generalization of this algorithm for complex queries.

We prove that this algorithm is competitive:

Theorem 2.2. *The online greedy query planning algorithm described above achieves a competitive ratio of* 4. *Further, if the QEPs satisfy the property that every pair of QEPs is sufficiently different in terms of total work, in particular,* $w_i \leq \frac{1}{2} w_{i-1}$ *for all* $i > 1$*, then the competitive ratio is* ≤ 2*, matching the lower bound.*

We note that constant competitive ratio implies that the performance of our online query planning algorithm is independent of the nature of workloads or the environment.

2.5 Implementation

Implementation of QOOP involved changes to Calcite [4], Hive [6], Tez [8], and YARN [109]. QOOP's implementation took $\sim 13k$ SLOC. The majority of our changes were in Tez mostly devoted to dynamic CBO module we elaborate upon shortly.

Hive and Calcite: Hive uses the Volcano query planner implemented in Calcite to get a cost-based optimized (CBO) plan. We add the ability to cache several logical plans in Calcite

during its plan evaluation process and make changes to Hive to fetch multiple physical plans (i.e., Tez QEPs). Also, we make changes to annotate each QEP with provenance—the set of logical relational operators associated with each stage of the QEP. We widened the RPC interface from Hive to Tez, to push multiple QEPs to Tez as part of a single job.

Tez and Yarn: To enable dynamic query plan switching we added modules to Tez that are responsible for (i) accounting checkpoints to enable backtracking; (ii) dynamic cost-based optimization to make Tez QEP switching decisions; (iii) runtime QEP changes to realize QEP switching; and (iv) the RPC mechanism from YARN to Tez to give resource feedback (i.e., resource updates about the dynamic "1-over-n" share of resources).

Any resource change event from YARN triggers our dynamic CBO module that evaluates all QEPs. This module first propagates progress using provenance and estimates completion time of each QEP via simulated packing in the available resource share (§2.3.4). Our CBO relies on estimates of tasks' resource demands—CPU, memory, disk, and the network—and their durations. Peak resource estimates are based on prior runs for each QEP. We use these peak resource estimates to decide the container request sizes for tasks in the currently executing QEP.)

Checkpointing for backtracking and runtime changes to the QEP involve changes to the QEP, Vertex, and Task state machines in Tez. All checkpointing state is maintained at the Tez QEPAppMaster—which keeps the file handle of task output after every task completion event. For QEP switching, we added the `SWITCHING` state to the QEP state machine. On a resource change event a QEP is forced from `RUNNING` to `SWITCHING`. Any running tasks of the QEP continue running in this state but the launch of any new vertex (and hence its tasks) is prevented in this state. The QEP switches to `RUNNING` state after, if at all, QEP switching happens. During a QEP switch, the set of runnable Vertices is re-initialized to those from the new QEP. The Vertex definition is changed so that the inputs for the tasks spawned by any runnable Vertex points to the appropriate checkpoint.

2.6 Evaluation

In this section, we evaluate QOOP in situations with varying degrees of resource variabilities. We examine both the performance of an individual query using QOOP's replanning as well as overall performance when multiple queries run atop QOOP.

We start by studying the execution of a single job, subjecting it to real resource change events or *resource profiles*. Specifically, in these *micro-benchmarks*, our focus is on answering the following key question: *does QOOP's dynamic query re-planning improve a job's completion time when compared to static, early-binding approaches?*

Next, we evaluate the key system components of QOOP – backtracking, overheads of QEP switching, robustness to errors in the task estimates, and hysteresis.

Finally, we consider a small private cluster, where QOOP is used to manage the execution of multiple jobs. We evaluate QOOP by running multiple jobs on the testbed, wherein job arrivals and completions can lead to large resource perturbations. This *macro-benchmark* addresses the question: *does QOOP's simple cluster scheduler and dynamic query re-planner approach improve system-wide objectives when compared against systems with complex schedulers and static query planners?*

2.6.1 Experimental Setup

Workloads: Our workloads consist of queries from the publicly available TPC-DS [18] benchmark. We experiment with a total of 50 queries running at a scale of 500, i.e., running on a 500GB dataset. [2] For micro-benchmarks, we focus on the perspectives of individual queries. For macro-benchmarks, each workload consists of jobs drawn at random from our 50 queries and arriving in a Poisson process with an average inter-arrival time of 80s. [3]

Cluster: Our testbed has 20 bare-metal servers – each machine has 32 cores, 128 GB of memory, 480 GB SSD, 1 Gbps NIC and runs Ubuntu 14.04. For micro-benchmarks, we evaluate QOOP under different realistic resource profiles, as elaborated later in this section. In such experiments, we provide as much resources from the cluster to each job over time as dictated by the resource profile. Specifically, whenever there is an increase in the amount of resources in the resource profile we make available to the job corresponding number of containers, whereas whenever there is a decrease in the amount of resources in the resource profile we immediately revoke equivalent number of containers and fail any tasks running on them.

For macro-benchmarks, we run our entire collection of jobs across the entire cluster. At its maximum capacity, the cluster can run 600 tasks (containers) in parallel.

Baselines: In micro-benchmarks, we compare QOOP's query planner against static query plans obtained from the *Clarinet QP*, which is a resource-aware QP implemented in Hive [111] that improves upon Volcano. We only compare against *Clarinet QP* as it outperforms Volcano. We adapted Clarinet to our setting to choose a QEP that minimizes completion time using resource estimates just before query execution begins. It represents the performance upper-bound of fixed-QEP approaches.

[2] We cached plans obtained while exploring QEPs in the Volcano planner, and retained plans with significant differences in cost according to Volcano's cost model. We used the first 50 TPC-DS queries that gave the most number of QEP alternatives.

[3] Google cluster trace [9] analysis on 20-machine sets yielded an average job inter-arrival time of 80s.

In macro-benchmarks, we compare QOOP – dynamic query planner on top of our simple max-min fair scheduler – against the following approaches on the three system-wide objectives of fairness, job completion time, and efficiency: (1) *DRF*: The default DRF multi-resource fair scheduler [48] in conjunction with Hive's default Volcano QP; (2) *Tetris*: A multi-resource packing scheduler [53] with Volcano; (3) *SJF*: Shortest-Job-First scheduler [45] with Volcano; (4) *Carbyne*: A meta-scheduler that leverages DRF, Tetris, and SJF [54] with Volcano; (5) *DRF+Clarinet*: DRF with the Clarinet QP [111]; (6) *Carbyne+Clarinet*: Carbyne scheduler with Clarinet QP.

These reflect combinations of query planners that differ in whether they are resource-aware with schedulers that differ in the complexity of information they leverage in making scheduling decisions.

Metrics: Our primary metric to quantify performance improvement using QOOP is improvement in the average *job completion time (JCT)*: $\frac{\text{(Average) JCT of an Approach}}{\text{(Average) JCT of QOOP}}$

Additionally, in multi-job scenarios, we consider *Jain's fairness index* [68] to measure fairness between jobs, and *makespan* (i.e., when the last job completes in a workload) to measure overall resource efficiency of the cluster.

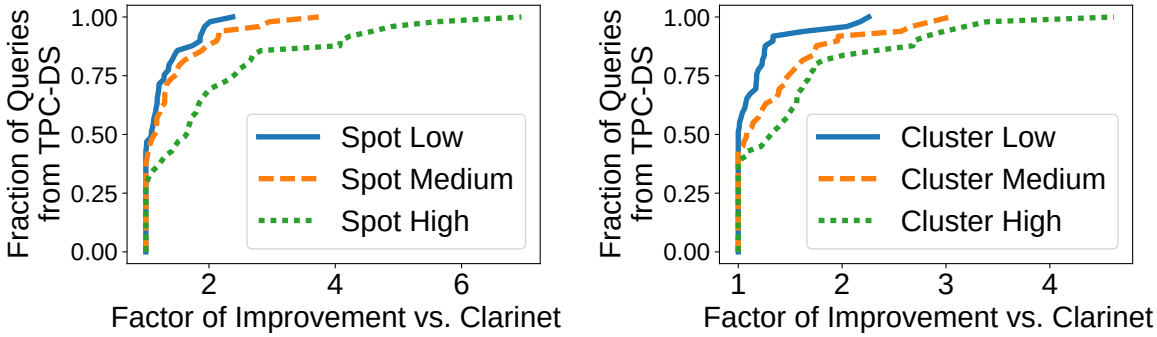
2.6.2 QOOP in Micro-Benchmarks

QOOP has two core components: the dynamic query replanning logic for a single query (Sections 2.3.3 and 2.3.4), and the simple cross-job cluster wide scheduler (Section 2.3.2). We study the two separately, with this section focusing on the former using micro-benchmarks.

Specifically, in these micro-benchmarks, we ask: given a certain resource change profile, how well does a single query perform from using QOOP's query replanning algorithms? We study QOOP under two classes of resource change profiles, *spot instances* and *cluster resources* .

2.6.2.1 Spot Markets Resource Profiles

We obtained a 5-hour spot market price trend for `i3.2xlarge` instance type in Amazon EC2 cluster in the `us-west-2c` region for the time period from 17:00 UTC to 21:00 UTC for September 21, 2017. We infer the resource profile for the spot market price trend by applying the bidding strategy described in Section 4.3. We then divide the entire resource profile into "low", "medium", and "high" regions by time. To do so, we divide the entire resource profile into 10 minute regions and calculate the maximum increase or decrease in the resources in this 10 minute region. We call an x% increase or decrease in resource in at least one of the resource dimensions (compute or memory) over some period of time as



(a) Spot market

(b) Shared cluster

Fig. 2.6 Improvements using QOOP w.r.t. Clarinet under resource variations observed in different resource profiles.

an x% resource volatility. If the maximum resource volatility in resources is less than 10% then we classify this region as "low", if it is between 10% and 20% then we classify this region as "medium", if it is greater than 20% then it is classified as "high". We also refer to these as having "low", "medium" and "high" resource volatility . We then run each of our 50 TPC-DS queries individually against each of these three resource profiles – "low", "medium", and "high" – using both QOOP and Clarinet. For each query run with a particular resource profile type, we pick 10 different randomly selected regions of that particular profile type and report the mean from these 10 runs.

We plot the CDF of QOOP's improvements over Clarinet for the three resource profiles in Figure 2.6a. We see that QOOP *strictly* outperforms Clarinet, with its gains improving with increasing resource volatility – overall, 58%, 62% and 66% of the jobs experience faster completion times in each of "low", "medium", and "high" profiles, respectively. Median improvements for the "low", "medium" and "high" profiles are respectively 1.08×, 1.11× and 1.47×. For "high" profiles, 10% of jobs see gains $> 4\times$! We also note that 34% of the jobs show no improvements over Clarinet even with "high". On further analysis, we found that these jobs are queries in the TPC-DS workload that are either (i) less complex queries with less number of joins, or (ii) queries with short durations. Less complex queries may lack attractive alternative QEPs, whereas short queries may miss out on resource perturbations. This limits opportunities for re-planning and improvement. We dig deeper into these issues later in this section.

2.6.2.2 Shared Cluster Resource Profile

Similar to the spot market scenario, we generate three different resource profiles for the shared cluster scenario described in Section 4.3. Following a similar methodology, we

identify “low”, “medium” and “high” resource volatility periods, and we run each of the 50 queries.

As before, we plot the CDF of QOOP’s improvements in Figure 2.6b. We see the trends similar to that of the spot market trace – overall, 56%, 58% and 60% of the jobs complete faster in “low”, “medium”, and “high” profiles, respectively. The median improvements in the three profiles are $1.08\times$, $1.11\times$ and $1.20\times$, with higher performance improvements in greater resource volatility scenarios; for the “high” profile, 10% of jobs see gains $> 3.3\times$.

In both the spot instance and cluster profiles, gains are higher for profiles with higher volatility. In other words, QOOP’s dynamic replanning is most effective relative to static query plans when resource volatility is at its highest. Also, the improvements for spot market and shared cluster, while similar for “low” and “medium”, differ on the “high” resource profiles. We attribute this to spot market “high” resource profiles experiencing 7% larger magnitudes of resource changes at median than that of the shared cluster.

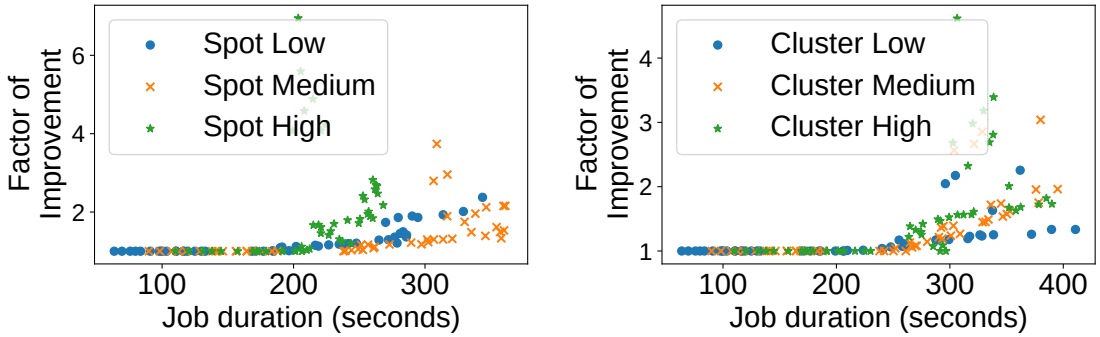
2.6.2.3 Delving into Improvements

Next, we take a deep dive into the aforementioned scenarios to understand when QOOP offers the greatest/least improvements. We study the impact of job duration, complexity, and the number of QEP switches that occur.

Job Durations vs. Observed Gains: The improvements in per-job performance due to QOOP as a function of job duration is shown in Figures 2.7a and 2.7b for the spot market and cluster resource profiles, respectively. Both figures also show results for the “low”, “medium” and “high” volatility profiles using different-colored dots. In both cases, QOOP’s benefits increase with increasing job durations. This is because longer jobs receive more opportunities for switching query plans and the comparative overhead of a switch of a longer job is smaller w.r.t. its completion time. Nevertheless, some shorter jobs benefit from QOOP in case of higher resource volatility.

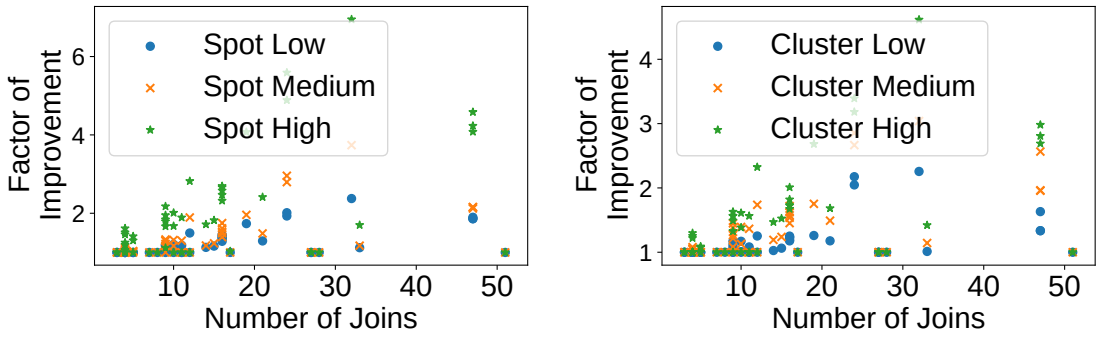
Job Complexity vs. Improvement: Figures 2.8a and 2.8b show improvements obtained with QOOP as we increase query complexity for the spot market and cluster profiles, respectively. We measure query complexity in terms of the number of join operations in the query. We make two observations. First, increased query complexity generally correlates with increased gains. This is because the number of alternate query execution plans is higher with a greater number of joins. Second, keeping complexity constant, higher volatility results in the highest factor of improvement (as indicated above).

QEP switches vs. Improvement: Figures 2.9a and 2.9b shows the trend between improvements and number of runtime QEP switches. First, we see that an increase in the number of query execution plan switches correlates with increased gains. Second, keeping the number



(a) Spot market (b) Shared cluster

Fig. 2.7 Improvements vs. job durations using QOOP w.r.t. Clarinet under different resource profiles.

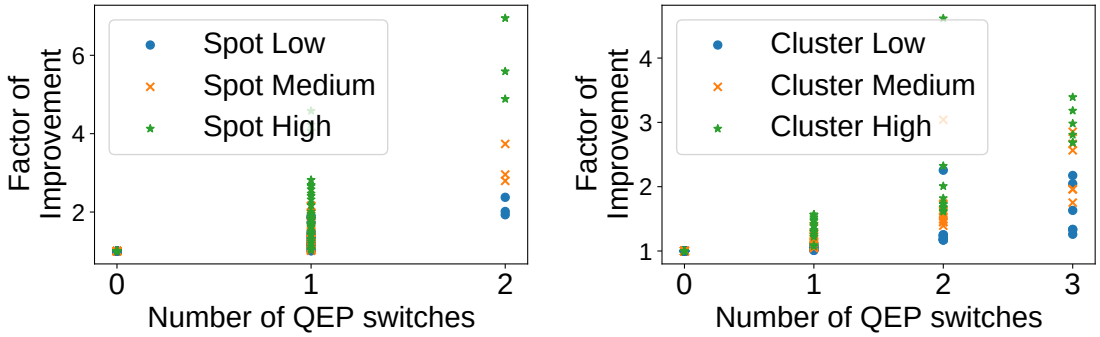


(a) Spot market (b) Shared cluster

Fig. 2.8 Improvements vs. query complexity (number of joins) using QOOP w.r.t. Clarinet under various resource profiles.

of switches constant, higher volatility results in the highest factor of improvement. In general, the greater flexibility a query intrinsically has in terms of multiple alternate plans together with the flexibility QOOP offers in switching to these plans results in a higher degree of improvement.

Task Throughput: Finally, we consider how fast QOOP helps the query complete tasks over time. We measure task throughput as the average number of tasks of the job executed per second; higher implies better utilization. In Figure 2.10 we show the task throughput of QOOP and Clarinet across queries. The number of tasks per second in the case of QOOP exceeds Clarinet by $\sim 24\%$ in the average case. Further analysis showed that an increase in the number of resources available leads QOOP to switch to query execution plans that favor more parallelism (i.e., "bushy" joins) and contributes to increased utilization.



(a) Spot market

(b) Shared cluster

Fig. 2.9 Improvements vs. number of QEP switches using QOOP w.r.t. Clarinet under various resource profiles.

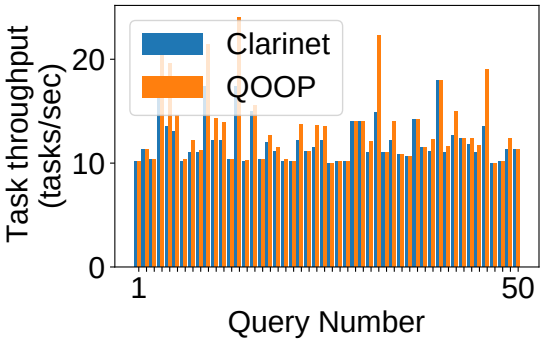


Fig. 2.10 Improvements w.r.t. Clarinet vs. number of QEP switches in spot market.

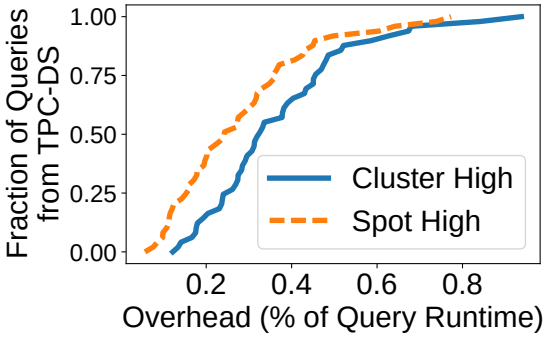


Fig. 2.11 Overheads due to QEP switches measured as % of a job's completion time.

2.6.3 Impact of Various QOOP Features

In this section, we study the effect of different aspects of QOOP on the performance observed by a single query.

Backtracking: Figure 2.12 shows the relationship between improvement factor and the depth of backtracking in a shared cluster setting with different resource profiles. We observe that the depth of backtracking (i.e. the maximum distance of the vertex in the switched-to QEP from any running/completed vertex in the current QEP) increases with the magnitude of resource change events. 5.7% of all the runs experience a backtracking to two stages deep in the past and is triggered only by "high" volatile resource profile. 85.3% of the experimental runs with "low" volatile resource profile experienced no backtracking. We observe similar results for spot market setting. Figure 2.13 shows the CDF of factor of improvement w.r.t. Clarinet with and without backtracking turned on for the runs of all our TPC-DS queries when run under shared cluster resource profiles. We observe that when

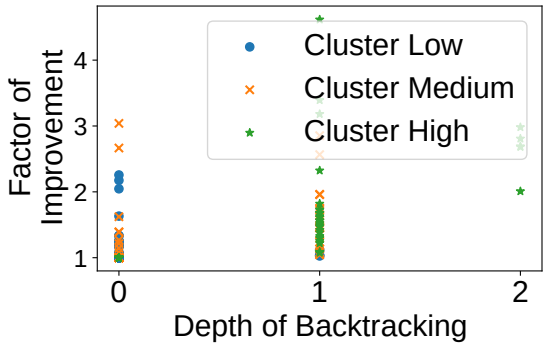


Fig. 2.12 Improvements vs. depth of backtracking.

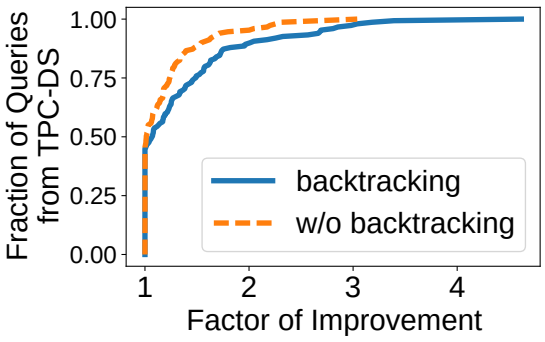


Fig. 2.13 Improvements with and without backtracking.

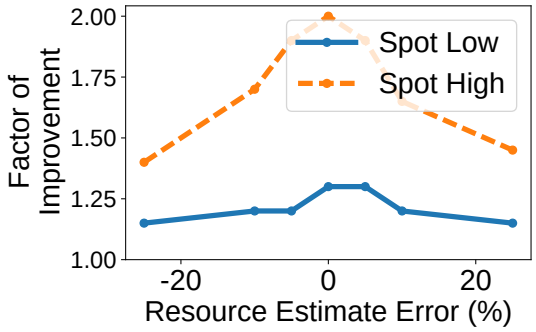


Fig. 2.14 Error robustness in QOOP.

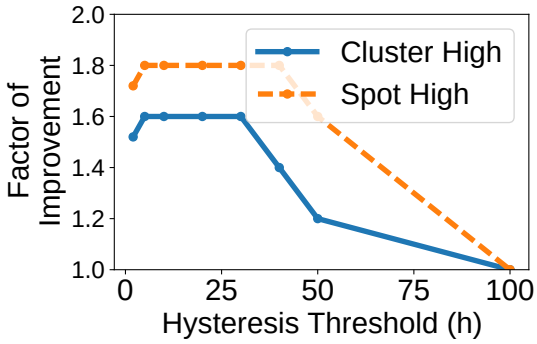


Fig. 2.15 Effect of hysteresis on improvements.

backtracking is turned on QOOP yields higher factor of improvement as backtracking finds better QEP switches.

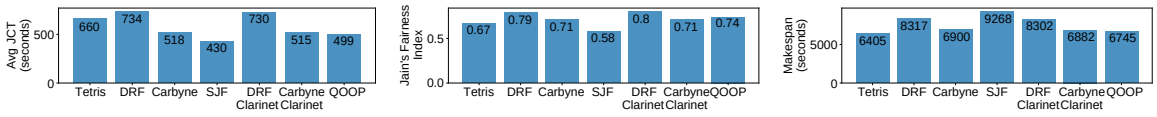
Overhead of QEP switch: Figure 2.11 shows the overheads of QOOP in the shared cluster and spot market settings. We measure overhead as the time a job spends in switching to alternate QEPs as a percentage of total job time. The overheads in the shared cluster are 0.15% higher than in the spot market setting. This is because of the higher number of overall QEP switches when a job runs in a shared cluster – also shown in Figure 2.9b. On the whole, however, the overhead due to QEP switching has negligible impact ($< 1\%$) on overall job performance. The overall QEP switching overhead is low as hysteresis prevents unnecessary QEP switching and the absolute number of QEP switches in a job is low – at most 3 as shown in Figures 2.9a and 2.9b.

Robustness to Error: Figure 2.14 shows QOOP's robustness to error in the estimates of task resource demands and durations. We introduce X% errors in our estimated task demands and durations. Specifically, we select X in [-25, 25] as suggested by prior work [54], and increase/decrease resource demands by $task_{newReq} = (1 + X/100) * task_{origReq}$, and task durations change similarly. We study these errors in simulation against low and high volatile

spot market resource profiles. We observe even at the highest error rates of $\pm 25\%$, QOOP offers substantial performance improvements (e.g., $1.4\times$ for the high volatile profile). For low volatile resource profile, QOOP is more robust to estimation errors: at 25% error rate, the performance improvement is $1.18\times$ compared to $1.25\times$ at no error. However, mis-estimations are costly at high volatility: errors $\geq 10\%$ cause performance improvement to drop 33% or more; nevertheless, QOOP's performance is always better than Clarinet.

Hysteresis: Figure 2.15 shows the effect of our hysteresis threshold (h) on the improvements. In QOOP, hysteresis prevents QEP switch unless there is an $h\%$ improvement in the estimated job completion time. We experiment with different values of h for "high" resource profiles for both spot market and shared cluster. A very high hysteresis threshold prevents switching, hurting performance. By definition, setting hysteresis parameter (h) to 0 causes more QEP switching (because of lower thresholds for QEP switching) and hence slightly higher overhead; we still see positive gains. However, for both traces, we observe that there is a wide range of h values where the factor of improvement sustains it's peak. This means that QOOP has flexibility to choose h; any value in the 10% - 25% range offers good performance at low switching overhead.

2.6.4 QOOP in Macro-Benchmarks



(a) Job Performance (b) Inter-Job Fairness (c) Cluster Efficiency

Fig. 2.16 Comparison of performance, fairness, and cluster efficiency of QOOP w.r.t. existing solutions. Higher values are better for fairness, whereas the opposite is true for the rest.

So far we have evaluated QOOP in offline, micro-benchmarks against the Clarinet QP with an aim to understand its query re-planning capabilities. In a real cluster, however, jobs arrive in an online fashion. Consequently, the impact of scheduling on job performance and its interplay with the QP become important.

In this section, we evaluate QOOP in an online setting in our shared cluster, where 200 TPC-DS jobs – randomly drawn from the 50 TPC-DS queries – arrive following a Poisson process with an average inter-arrival time of 80 seconds (Figure 2.16). As mentioned earlier, we compare QOOP against a wide range of solutions in both categories: scheduling and query planning. On the one hand, we consider a variety of scheduling solutions such as DRF, Tetris, SJF, and Carbyne that focus on objectives ranging from simple fairness (QOOP)

to improving multiple goals. On the other hand, we consider QPs that range from static resource-agnostic planning (Volcano in Hive) to resource-aware early-binding (Clarinet) to QOOP's late-binding re-planner. Finally, in addition to focusing only on job completion time, which is useful only to individual jobs, we consider cluster-level metrics such as fairness (measured in terms of Jain's fairness index [68]) and efficiency (measured in terms of makespan).

Job Performance: First, we observe that QOOP significantly improves the average JCT w.r.t. simple state-of-the-art solutions (Tetris, DRF) and comes closest to the average JCT of SJF (Figure 2.16a). Furthermore, it outperforms the state-of-the-art in complex scheduling and QP alternatives: Carbyne and DRF+Clarinet, respectively. Only by combining two complicated solutions (Carbyne+Clarinet), the state-of-the-art can come close to QOOP. This suggests that the inflexibility of the current interfaces have tangible costs and overcoming them requires introducing complexities at every layer of the analytics stack.

Fairness Between Jobs: If performance were the only concern, one could get away with simply using SJF instead of using the complex alternatives or QOOP. However, performance and fairness have a strong tradeoff [54] as shown in Figure 2.16b – SJF has the worst fairness characteristics! We observe that while DRF and DRF+Clarinet are the most fair solutions, QOOP comes the closest to them while ensuring almost 1.5$\times$ smaller average JCT.

Cluster Efficiency: Finally, Tetris performs well in its goal of packing tasks better and achieving high efficiency (Figure 2.16c), but QOOP again comes the closest to Tetris.

Overall, QOOP improves all three metrics – performance, fairness, and efficiency – over complex state-of-the-art solutions or combinations thereof, and achieves these benefits using a simple scheduler with a dynamic, resource-aware QP that can re-plan queries at runtime.

2.7 Related Work

Other Applications: Although we focus on SQL queries, the high-level principle of designing dynamic resource-aware plan switching can be applied to many other applications. This is because many frameworks use query planners to create execution plans for workloads, e.g., in machine learning [50, 74, 83], graph processing [52, 79, 81], approximation [22, 14] and streaming [123, 17, 19, 23, 87].

Query Planners in Big Data Clusters: Query planning is a well-trodden research area with numerous prior work [62]. We restrict our focus on query planners designed for distributed big data clusters that fall into two broad categories: those that plan a query in a resource-agnostic manner [4, 27] and those that are resource-aware [111]. Both, however, result in static query plans throughout the execution of a job. There is a massive body of work

on adaptive query processing [43] in the context of traditional (single-machine) database systems. We focus on big data analytics in multi-node clusters.

Execution Engines: Execution engines take job DAGs and interact with the cluster scheduler to run all the tasks of each job until its completion. Examples of popular execution engines include Apache Spark [122], Dryad [65, 120], and Apache Tez [8]. Execution engines such as Tez [8] and DryadLINQ [120] allow for dynamic optimizations to the job DAG in the form of dynamism in vertex parallelism, data partitioning, and aggregation tree but lack the interfaces to make logical-level DAG switches.

Cluster Schedulers: Today's schedulers are multi-resource [48, 53, 73, 36, 24], DAG-aware [35, 53, 122], and allow a variety of constraints [124, 66, 25, 49, 121]. Given all these inputs, they optimize for objectives such as fairness [48, 67, 47, 29], performance [45], efficiency [53], or different combinations of the three [54, 55]. Over time, schedulers are becoming more complex and taking increasingly more job-level information as inputs. In contrast, we propose a simplified scheduler and argue for pushing complexity up the stack.

Chapter 3

THEMIS: Fair and Efficient GPU Cluster Scheduling

3.1 Introduction

With the widespread success of machine learning (ML) for tasks such as object detection, speech recognition, and machine translation, a number of enterprises are now incorporating ML models into their products. Training individual ML models is time- and resource-intensive with each training job typically executing in parallel on a number of GPUs.

With different groups in the same organization training ML models, it is beneficial to consolidate GPU resources into a shared cluster. Similar to existing clusters used for large scale data analytics, shared GPU clusters for ML have a number of operational advantages, e.g., reduced development overheads, lower costs for maintaining GPUs, etc. However, today, there are no ML workload-specific mechanisms to share a GPU cluster in a *fair* manner.

Our conversations with cluster operators indicate that fairness is crucial; specifically, that sharing an ML cluster becomes attractive to users only if they have the appropriate *sharing incentive*. That is, if there are a total N users sharing a cluster C, every user's performance should be no worse than using a private cluster of size $\frac{C}{N}$. Absent such incentive, users are either forced to sacrifice performance and suffer long wait times for getting their ML jobs scheduled, or abandon shared clusters and deploy their own expensive hardware.

Providing sharing incentive through fair scheduling mechanisms has been widely studied in prior cluster scheduling frameworks, e.g., Quincy [66], DRF [48], and Carbyne [54]. However, these techniques were designed for big data workloads, and while they are used widely to manage GPU clusters today, they are far from effective.

The key reason is that ML workloads have unique characteristics that make existing "fair" allocation schemes actually *unfair*. First, unlike batch analytics workloads, ML jobs have *long running tasks* that need to be scheduled together, i.e., gang-scheduled. Second, each task in a job often runs for a number of iterations while synchronizing model updates at the end of each iteration. This frequent communication means that jobs are *placement-sensitive*, i.e., placing all the tasks for a job on the same machine or the same rack can lead to significant speedups. Equally importantly, as we show, ML jobs differ in their placement-sensitivity (Section 3.3.1.2).

In Section 3.3, we show that having long-running tasks means that established schemes such as DRF – which aims to equally allocate the GPUs released upon task completions – can arbitrarily violate sharing incentive. We show that even if GPU resources were released/reallocated on fine time-scales [56], placement sensitivity means that jobs with same aggregate resources could have widely different performance, violating sharing incentive. Finally, heterogeneity in placement sensitivity means that existing scheduling schemes *also violate* Pareto efficiency and envy-freedom, two other properties that are central to fairness [107].

Our scheduler, THEMIS, address these challenges, and supports sharing incentive, Pareto efficiency, and envy-freedom for ML workloads. It multiplexes a GPU cluster across *ML applications* (Section 4.3), or apps for short, where every app consists of one or more related ML jobs, each running with different hyper-parameters, to train an accurate model for a given task. To capture the effect of long running tasks and placement sensitivity, THEMIS uses a new long-term fairness metric, *finish-time fairness*, which is the ratio of the running time in a shared cluster with N apps to running alone in a $\frac{1}{N}$ cluster. THEMIS's goal is thus to minimize the maximum finish time fairness across all ML apps while efficiently utilizing cluster GPUs. We achieve this goal using two key ideas.

First, we propose to widen the API between ML apps and the scheduler to allow apps to specify placement preferences. We do this by introducing the notion of a round-by-round auction. THEMIS uses leases to account for long-running ML tasks, and auction rounds start when leases expire. At the start of a round, our scheduler requests apps for their finish-time fairness metrics, and makes all available GPUs visible to a fraction of apps that are currently farthest in terms of their fairness metric. Each such app has the opportunity to *bid* for subsets of these GPUs as a part of an auction; bid values reflect the app's new (placement sensitive) finish time fairness metric from acquiring different GPU subsets. A central arbiter determines the global winning bids to maximize the aggregate improvement in the finish time fair metrics across all bidding apps. Using auctions means that we need to ensure that apps are truthful when they bid for GPUs. Thus, we use a *partial allocation* auction that incentivizes truth telling, and ensures Pareto-efficiency and envy-freeness by design.

While a far-from-fair app may lose an auction round, perhaps because it is placed less ideally than another app, its bid values for subsequent auctions naturally increase (because a losing app's finish time fairness worsens), thereby improving the odds of it winning future rounds. Thus, our approach converges to fair allocations over the long term, while staying efficient and placement-sensitive in the short term.

Second, we present a two-level scheduling design that contains a centralized inter-app scheduler at the bottom level, and a narrow API to integrate with existing hyper-parameter tuning frameworks at the top level. A number of existing frameworks such as Hyperdrive [98] and HyperOpt [28] can intelligently apportion GPU resources between various jobs in a single app, and in some cases also terminate a job early if its progress is not promising. Our design allows apps to directly use such existing hyper parameter tuning frameworks. We describe how THEMIS accommodates various hyper-parameter tuning systems and how its API is exercised in extracting relevant inputs from apps when running auctions.

We implement THEMIS atop Apache YARN 3.2.0, and evaluate by replaying workloads from a large enterprise trace. Our results show that THEMIS is at least 2.25X more fair (finish-time fair) than state-of-the-art schedulers while also improving cluster efficiency by ~5% to 250%. To further understand our scheduling decisions, we perform an event-driven simulation using the same trace, and our results show that THEMIS offers greater benefits when we increase the fraction of network intensive apps, and the cluster contention.

3.2 Motivation

We start by defining the terminology used in the rest of the work. We then study the unique properties of ML workload traces from a ML training GPU cluster at Microsoft. We end by stating our goals based on our trace analysis and conversations with the cluster operators.

3.2.1 Preliminaries

We define an ML app, or simply an "app", as a collection of one or more ML model *training* jobs. Each app corresponds to a user training an ML model for a high-level goal, such as speech recognition or object detection. Users train these models knowing the appropriate hyper-parameters (in which case there is just a single job in the app), or they train a closely related set of models (n jobs) that explore hyper-parameters such as learning rate, momentum etc. [98, 77] to identify and train the best target model for the activity at hand.

Each job's constituent work is performed by a number of parallel *tasks*. At any given time, all of a job's tasks collectively process a *mini-batch* of training data; we assume that

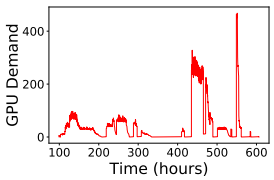


Fig. 3.1 Aggregate GPU demand of ML apps over time

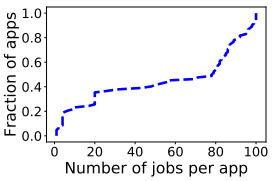


Fig. 3.2 Job count distribution across different apps

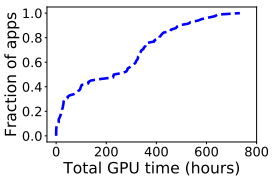


Fig. 3.3 ML app time (= total GPU time across all jobs in app) distribution

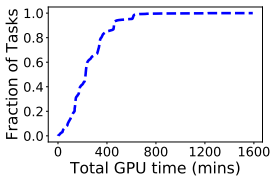


Fig. 3.4 Distribution of Task GPU times

the size of the batch is fixed for the duration of a job. Each task typically processes a subset of the batch, and, starting from an initial version of the model, executes multiple iterations of the underlying learning algorithm to improve the model. We assume all jobs use the popular synchronous SGD [33].

We consider the finish time of an app to be when the best model and relevant hyper-parameters have been identified. Along the course of identifying such a model, the app may decide to terminate some of its constituent jobs early [98, 28]; such jobs may be exploring hyper-parameters that are clearly sub-optimal (the jobs' validation accuracy improvement over iterations is significantly worse than other jobs in the same app). For apps that contain a single job, finish time is the time taken to train this model to a target accuracy or maximum number of iterations.

3.2.2 Characterizing Production ML Apps

We perform an analysis of the properties of GPU-based ML training workloads by analyzing workload traces obtained from Microsoft. The GPU cluster we study supports over 5000 unique users. We restrict our analysis to a subset of the trace that contains 85 ML training apps submitted using a hyper-parameter tuning framework.

GPU clusters are known to be heavily contented [70], and we find this also holds true in the subset of the trace of ML apps we consider (Figure 3.1). For instance, we see that GPU demand is bursty and the average GPU demand is ~50 GPUs.

We also use the trace to provide a first-of-a-kind view into the characteristics of ML apps. As mentioned in Section 3.2.1, apps may either train a single model to reach a target accuracy (1 job) or may use the cluster to explore various hyper-parameters for a given model (n jobs). Figure 3.2 shows that ~10% of the apps have 1 job, and around ~90% of the apps perform hyper-parameter exploration with as many as 100 jobs (median of 75 jobs). Interestingly, there is also a significant variation in the number of hyper-parameters explored ranging from a few tens to about a hundred (not shown).

We also measure the *GPU time* of all ML apps in the trace. If an app uses 2 jobs with 2 GPUs each for a period of 10 minutes, then the GPU time for — the tasks would be 10 minutes each, the jobs would be 20 minutes each, and the app would be 40 GPU minutes. Figure 3.3 and Figure 3.4 show the long running nature of ML apps: the median app takes 11.5 GPU days and the median task takes 3.75 GPU hours. There is a wide diversity with a significant fraction of jobs and apps that are more than 10X shorter and many that are more than 10X longer.

From our analysis we see that ML apps are heterogeneous in terms of resource usage, and number of jobs submitted. Running times are also heterogeneous, but at the same time much longer than, e.g., running times of big data analytics jobs (typically a few hours [55]). Handling such heterogeneity can be challenging for scheduling frameworks, and the long running nature may make controlling app performance particularly difficult in a shared setting with high contention.

We next discuss how some of these challenges manifest in practice from both cluster user and cluster operator perspectives, and how that leads to our design goals for THEMIS.

3.2.3 Our Goal

Our many conversations with operators of GPU clusters revealed a common sentiment, reflected in the following quote:

" We were scheduling with a balanced approach ... with guidance to 'play nice'. Without firm guard rails, however, there were always individuals who would ignore the rules and dominate the capacity. "

— An operator of a large GPU cluster at Microsoft

With long app durations, users who dominate capacity impose high waiting times on many other users. Some such users are forced to "quit" the cluster as reflected in this quote:

"Even with existing fair sharing schemes, we do find users frustrated with the inability to get their work done in a timely way... The frustration frequently reaches the point where groups attempt or succeed at buying their own hardware tailored to their needs. "

— An operator of a large GPU cluster at Microsoft

While it is important to design a cluster scheduler that ensures efficient use of highly contended GPU resources, the above indicates that it is perhaps equally, if not more important, for the scheduler to allocate GPU resources in a fair manner across many diverse ML apps; in other words, roughly speaking, the scheduler's goal should be to allow all apps to execute their work in a "timely way".

In what follows, we explain using examples, measurements, and analysis, why existing fair sharing approaches when applied to ML clusters fall short of the above goal, which we

formalize next. We identify the need both for a new fairness metric, and for a new scheduler architecture and API that supports resource division according to the metric.

3.3 Finish-Time Fair Allocation

We present additional unique attributes of ML apps and discuss how they, and the above attributes, affect existing fair sharing schemes.

3.3.1 Fair Sharing Concerns for ML Apps

The central question is - given R GPUs in a cluster C and N ML apps, what is a *fair way* to divide the GPUs.

As mentioned above, cluster operators indicate that the primary concern for users sharing an ML cluster is performance isolation that results in "timely completion". We formalize this as: if N ML Apps are sharing a cluster then an app should not run slower on the shared cluster compared to a dedicated cluster with $\frac{1}{N}$ of the resources. Similar to prior work [48], we refer to this property as *sharing incentive* (SI). Ensuring sharing incentive for ML apps is our primary design goal.

In addition, resource allocation mechanisms must satisfy two other basic properties that are central to fairness [107]: Pareto Efficiency (PE) and Envy-Freeness (EF) [1]

While prior systems like Quincy [66], DRF [48] etc. aim at providing SI, PE and EF, we find that they are ineffective for ML clusters as they fail to consider *the long durations of ML tasks* and *placement preferences of ML apps*.

3.3.1.1 ML Task Durations

We empirically study task durations in ML apps and show how they affect the applicability of existing fair sharing schemes.

Figure 3.4 shows the distribution of task durations for ML apps in a large GPU cluster at Microsoft. We note that the tasks are, in general, very long, with the median task roughly 3.75 hours long. This is in stark contrast with, e.g., big data analytics jobs, where tasks are typically much shorter in duration [92].

State of the art fair allocation schemes such as DRF [48] provide instantaneous resource fairness. Whenever resources become available, they are allocated to the task from an app with the least current share. For big data analytics, where task durations are short,

[1] Informally, a Pareto Efficient allocation is one where no app's allocation can be improved without hurting some other app. And, envy-freeness means that no app should prefer the resource allocation of an other app.

	VGG16	Inception-v3
4 P100 GPUs on 1 server	103.6 images/sec	242 images/sec
4 P100 GPUs across 2 servers	80.4 images/sec	243 images/sec

Table 3.1 Effect of GPU resource allocation on job throughput. VGG16 has a machine-local task placement preference while Inception-v3 does not.

this approximates instantaneous resource fairness, as frequent task completions serve as opportunities to redistribute resources. However, blindly applying such schemes to ML apps can be disastrous: running the much longer-duration ML tasks to completion could lead to newly arriving jobs waiting inordinately long for resources. This leads to violation of SI for late-arriving jobs.

Recent "attained-service" based schemes address this problem with DRF. In [56], for example, GPUs are leased for a certain duration, and when leases expire, available GPUs are given to the job that received the least GPU time thus far; this is the "least attained service", or LAS allocation policy. While this scheme avoids the starvation problem above for late-arriving jobs, it still violates all key fairness properties because it is placement-unaware, an issue we discuss next.

3.3.1.2 Placement Preferences

Next, we empirically study placement preferences of ML apps. We use examples to show how ignoring these preferences in fair sharing schemes violates key properties of fairness.

Many apps, many preference patterns: ML cluster users today train a variety of ML apps across domains like computer vision, NLP and speech recognition. These models have significantly different model architectures, and more importantly, different placement preferences arising from different computation, communication needs. For example, as shown in Table 3.1, VGG16 has a strict machine-local task placement preference while Inception-v3 does not. This preference inherently stems from the fact that VGG-like architectures have very large number of parameters and incur greater overheads for updating gradients over the network.

We use examples to show the effect of placement on DRF's allocation strategy. Similar examples and conclusions apply for the LAS allocation scheme.

Ignoring placement affects SI: example: Consider the Instance 1 in Figure 3.5. In this example, there are two placement sensitive ML apps - A_1 and A_2, both training VGG16. Each ML app has just one job in it with 4 tasks and the cluster has two 4 GPU machines. As shown above, given the same number of GPUs both apps prefer GPUs to be in the same server than spread across servers.

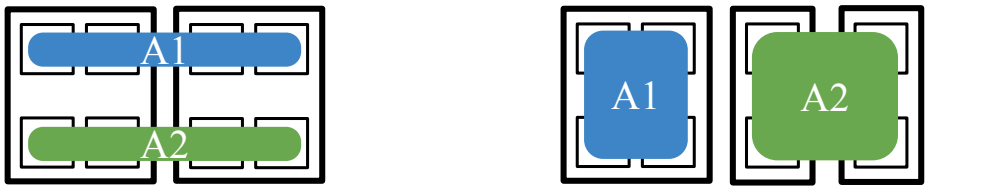


Fig. 3.5 By ignoring placement preference, DRF violates sharing incentive.

For this example, DRF [48] equalizes the dominant resource share of both the apps under resource constraints and allocates 4 GPUs to each ML app. In Instance 1 of Figure 3.5 we show an example of a valid DRF allocation. Both apps get the same type of placement with GPUs spread across servers. This allocation violates SI for both apps as their performance would be better if each app just had its own dedicated server.

Ignoring placement affects PE, EF: example: Consider Instance 2 in Figure 3.5 with two apps - A_1 (Inception-v3) which is not placement sensitive and A_2 (VGG16) which is placement sensitive. Each app has one job with four tasks and the cluster has two machines: one 4 GPU and two 2 GPU.

Now consider the allocation in Instance 2, where A_1 is allocated on the 4 GPU machine whereas A_2 is allocated across the 2 GPU machines. This allocation violates EF, because A_2 would prefer A_1's allocation. It also violates PE because swapping the two apps' allocation would improve A_2's performance without hurting A_1.

In fact, we can formally show that:

Theorem 3.1. *Existing fair schemes (DRF, LAS) ignore placement preferences and violate SI, PE, EF for ML apps.*

Proof Refer to Appendix.

In summary, existing schemes fail to provide fair sharing guarantees as they are unaware of ML app characteristics. Instantaneous fair schemes such as DRF fail to account for long task durations. While least-attained service schemes overcome that limitation, neither approach's input encodes placement preferences. Correspondingly, the fairness metrics used - i.e., dominant resource share (DRF) or attained service (LAS) - do not capture placement preferences.

This motivates the need for a new placement-aware fairness metric, and corresponding scheduling discipline. Our observations about ML task durations imply that, like LAS, our fair allocation discipline should not depend on rapid task completions, but instead should operate over longer time scales.

$\overrightarrow{G}$	$[0,0]$	$[0,1]=[1,0]$	$[1,1]$
ρ	ρ_{old}	$\frac{200}{400}=\frac{1}{2}$	$\frac{100}{400}=\frac{1}{4}$

Table 3.2 Example table of bids sent from apps to the scheduler

3.3.2 Metric: Finish-Time Fairness

We propose a new metric called as finish-time fairness, ρ. $\rho = \frac{T_{sh}}{T_{id}}$.

T_{id} is the *independent finish-time* and T_{sh} is the *shared finish-time*. T_{sh} is the finish-time of the app in the shared cluster and it encompasses the slowdown due to the placement and any queuing delays that an app experiences in getting scheduled in the shared cluster. The worse the placement, the higher is the value of T_{sh}. T_{id}, is the finish-time of the ML app in its own independent and exclusive $\frac{1}{N}$ share of the cluster. Given the above definition, sharing incentive for an ML app can be attained if $\rho \leq 1$. [2]

To ensure this, it is necessary for the allocation mechanism to estimate the values of ρ for different GPU allocations. Given the difficulty in predicting how various apps will react to different allocations, it is intractable for the scheduling engine to predict or determine the values of ρ.

Thus, we propose a new wider interface between the app and the scheduling engine that can allow the app to express a *preference* for each allocation. We propose that apps can encode this information as a table. In Table 3.2, each column has a permutation of a potential GPU allocation and the estimate of ρ on receiving this allocation. We next describe how the scheduling engine can use this to provide fair allocations.

3.3.3 Mechanism: Partial Allocation Auctions

The finish-time fairness $\rho_i(.)$ for an ML app A_i is a function of the GPU allocation $\vec{G_i}$ that it receives. The allocation policy takes these $\rho_i(.)$'s as inputs and outputs allocations $\vec{G_i}$.

A straw-man policy that sorts apps based on their reported ρ_i values and allocates GPUs in that order reduces the maximum value of ρ but has one key issue. An app can submit *false information* about their ρ values. This greedy behavior can boost their chance of winning allocations. Our conversations with cluster operators indicate that apps request for more resources than required and they require manual monitoring ("*We also monitor the usage. If they don't use it, we reclaim it and pass it on to the next approved project*"). Thus, this

[2]Note, sharing incentive criteria of $\rho \leq 1$ assumes the presence of an admission control mechanism to limit contention for GPU resources. An admission control mechanism that rejects any app if the aggregate number of GPUs requested crosses a certain threshold is a reasonable choice.

Pseudocode 3 Finish-Time Fair Policy

1: Applications $\{A_i\}$ ▷ set of apps
2: Bids $\{\lesssim_{i}(.)\}$ ▷ valuation function for each app i
3: Resources $\overrightarrow{R}$ ▷ resource set available for auction
4: Resource Allocations $\{\overrightarrow{G}_i\}$ ▷ resource allocation for each app i

5: **procedure** AUCTION($\{A_i\}$, $\{\lesssim_{i}(.)\}$, $\overrightarrow{R}$)
6: $\overrightarrow{G}_{i,pf}$ = arg max $\prod_i 1/\lesssim_{i}(\overrightarrow{G_i})$ ▷ proportional fair (pf) allocation per app i
7: $\overrightarrow{G}^{-i}_{j,pf}$ = arg max $\prod_{j!=i} 1/\lesssim_{j}(\overrightarrow{G_j})$ ▷ pf allocation per app j without app i
8: $c_i = \frac{\prod_{j!=i} 1/\lesssim_{j}(\overrightarrow{G}_{j,pf})}{\prod_{j!=i} 1/\lesssim_{j}(\overrightarrow{G}^{-i}_{j,pf})}$
9: $\overrightarrow{G_i} = c_i * \overrightarrow{G}_{i,pf}$ ▷ allocation per app i
10: $\overrightarrow{L} = \sum_i 1 - c_i * \overrightarrow{G}_{i,pf}$ ▷ aggregate leftover resource
11: return $\{\overrightarrow{G_i}\}$, $\overrightarrow{L}$
12: **procedure** ROUNDBYROUNDAUCTIONS($\{A_i\}$, $\{\lesssim_{i}(.)\}$)
13: **while** True **do**
14: ONRESOURCEAVAILABLEEVENT $\vec{R'}$:
15: $\{A_i^{sort}\}$ = SORT($\{A_i\}$) on $\rho_i^{current}$
16: $\{A_i^{filter}\}$ = get top $1-f$ fraction of apps from $\{A_{sort}\}$
17: $\{\rho_i^{filter}(.)\}$ = get updated $\rho(.)$ from apps in $\{A_i^{filter}\}$
18: $\{\overrightarrow{G_i^{filter}}\}$, $\overrightarrow{L}$ = AUCTION($\{A_i^{filter}\}$, $\{\rho_i^{filter}(.)\}$, $\vec{R'}$)
19: $\{A_i^{unfilter}\} = \{A_i\} - \{A_i^{filter}\}$
20: allocate $\overrightarrow{L}$ to $\{A_i^{unfilter}\}$ at random

simple straw-man fails to incentivize truth-telling and violates another key property, namely, *strategy proofness* (SP).

To address this challenge, we propose to use *auctions* in THEMIS. We begin by describing a simple mechanism that runs a single-round auction and then extend to a round-by-round mechanism that also considers online updates.

3.3.3.1 One-Shot Auction

Details of the inputs necessary to run the auction are given first, followed by how the auction works given these inputs.

Inputs: Resources and Bids. $\vec{R}$ represents the total GPU resources to be auctioned, where each element is 1 and the number of dimensions is the number of GPUs to be auctioned.

Each ML app bids for these resources. The bid for each ML app consists of the estimated finish-time fair metric (ρ_i) values for several different GPU allocations ($\vec{G_i}$). Each element in $\vec{G_i}$ can be $\{0, 1\}$. A set bit implies that GPU is allocated to the app. Example of a bid can be seen in Table 3.2.

Auction Overview. To ensure that the auction can provide strategy proofness, we propose using a *partial allocation* auction (PA) mechanism [38]. Partial allocation auctions have been shown to incentivize truth telling and are an appropriate fit for modeling subsets of indivisible goods to be auctioned across apps. Pseudocode 3, line 5 shows the PA mechanism. There are two aspects to auctions that are described next.

1. Initial allocation. PA starts by calculating an intrinsically proportionally-fair allocation $\vec{G_{i,pf}}$ for each app A_i by maximizing the product of the valuation functions i.e., the finish-time fair metric values for all apps (Pseudocode 3, line 6). Such an allocation ensures that it is not possible to increase the allocation of an app without decreasing the allocation of at least another app (satisfying PE [38]).
2. Incentivizing Truth Telling. To induce truthful reporting of the bids, the PA mechanism allocates app A_i only a fraction $c_i < 1$ of A_i's proportional fair allocation $\vec{G_{i,pf}}$, and takes $1 - c_i$ as a *hidden payment* (Pseudocode 3, line 10). The c_i is directly proportional to the decrease in the collective valuation of the other bidding apps in a market with and without app A_i (Pseudocode 3, line 8). This yields the final allocation $\vec{G_i}$ for app A_i (Pseudocode 3, line 9).

Note that the final result, $\vec{G_i}$ is not a market-clearing allocation and there could be unallocated GPUs $\vec{L}$ that are leftover from hidden payments. Hence, PA is not work-conserving. Thus, while the one-shot auction provides a number of properties related to fair sharing it does not ensure SI is met.

Theorem 3.2. *The one-shot partial allocation auction guarantees SP, PE and EF, but does not provide SI.*

Proof Refer to Appendix. The intuitive reason for this is that, with unallocated GPUs as hidden payments, PA does not guarantee $\rho \leq 1$ for all apps. To address this we next look at multi-round auctions that can maximize SI for ML apps. We design a mechanism that is based on PA and preserves its properties, but offers slightly weaker guarantee, namely min max ρ. We describe this next. It runs in multiple rounds. Empirically, we find that it gets $\rho \leq 1$ for most apps, even without admission control.

3.3.3.2 Multi-round auctions

To maximize sharing incentive and to ensure work conservation, our goal is to ensure $\rho \leq 1$ for as many apps as possible. We do this using three key ideas described below.
Round-by-Round Auctions: With round-by-round auctions, the outcome of an allocation from an auction is binding only for a *lease* duration. At the end of this lease, the freed GPUs are re-auctioned. This also handles the online case as any auction is triggered on a *resource available event*. This takes care of app failures and arrivals, as well as cluster reconfigurations.

At the beginning of each round of auction, the policy solicits updated valuation functions $\rho(.)$ from the apps. The estimated work and the placement preferences for the case of ML apps are typically time varying. This also makes our policy adaptive to such changes.

Round-by-Round Filtering: To maximize the number of apps with $\rho \leq 1$, at the beginning of each round of auctions we filter the $1-f$ fraction of total active apps with the greatest values of current estimate of their finish-time fair metric ρ. Here, $f \in (0,1)$ is a system-wide parameter.

This has the effect of restricting the auctions to the apps that are at risk of not meeting SI. Also, this restricts the auction to a smaller set of apps which reduces contention for resources and hence results in smaller hidden payments. It also makes the auction computationally tractable.

Over the course of many rounds, filtering maximizes the number of apps that have SI. Consider a far-from-fair app i that lost an auction round. It will appear in future rounds with much greater likelihood relative to another less far-from-fair app k that won the auction round. This is because, the winning app k was allocated resources; as a result, it will see its ρ improve over time; thus, it will eventually not appear in the fraction $1-f$ of not-so-fairly-treated apps that participate in future rounds. In contrast, i's ρ will increase due to the waiting time, and thus it will continue to appear in future rounds. Further an app that loses multiple rounds will eventually lose its lease on all resources and make no further progress, causing its ρ to become unbounded. The next auction round the app participates in will likely see the app's bid winning, because any non-zero GPU allocation to that app will lead to a huge improvement in the app's valuation.

As $f \to 1$, our policy provides greater guarantee on SI. However, this increase in SI comes at the cost of efficiency. This is because $f \to 1$ restricts the set of apps to which available GPUs will be allocated; with $f \to 0$ available GPUs can be allocated to apps that benefit most from better placement, which improves efficiency at the risk of violating SI.

Leftover Allocation: At the end of each round we have leftover GPUs due to hidden payments. We allocate these GPUs at random to the apps that did not participate in the auction in this round. Thus our overall scheme is work-conserving.

Overall, we prove that:

Theorem 3.3. *Round-by-round auctions preserve the PE, EF and SP properties of partial auctions and maximize SI.*

*P*roof. Refer to Appendix.

To summarize, in THEMIS we propose a new finish-time fairness metric that captures fairness for long-running, placement sensitive ML apps. To perform allocations, we propose using a multi-round partial allocation auction that incentivizes truth telling and provides Pareto efficient, envy free allocations. By filtering the apps considered in the auction, we maximize sharing incentive and hence satisfy all the properties necessary for fair sharing among ML applications.

3.4 System Design

We first list design requirements for an ML cluster scheduler taking into account the fairness metric and auction mechanism described in Section 3.3, and the implications for the THEMIS scheduler architecture. Then, we discuss the *API* between the scheduler and the hyper-parameter optimizers.

3.4.1 Design Requirements

Separation of visibility and allocation of resources. Core to our partial allocation mechanism is the abstraction of making available resources visible to a number of apps but allocating each resource exclusively to a single app. As we argue below, existing scheduling architectures couple these concerns and thus necessitate the design of a new scheduler.
Integration with hyper-parameter tuning systems. Hyper-parameter optimization systems such as Hyperband [77], Hyperdrive [98] have their own schedulers that decide the resource allocation and execution schedule for the jobs within those apps. We refer to these as app-schedulers. One of our goals in THEMIS is to integrate with these systems with minimal modifications to app-schedulers.

These two requirements guide our design of a new *two-level semi-optimistic* scheduler and a set of corresponding abstractions to support hyper-parameter tuning systems.

3.4.2 THEMIS Scheduler Architecture

Existing scheduler architectures are either pessimistic or fully optimistic and both these approaches are not suitable for realizing multi-round auctions. We first describe their shortcomings and then describe our proposed architecture.

3.4.2.1 Need for a new scheduling architecture

Two-level pessimistic schedulers like Mesos [63] enforce pessimistic concurrency control. This means that visibility and allocation go hand-in-hand at the granularity of a single app. There is restricted single-app visibility as available resources are partitioned by a mechanism internal to the lower-level (i.e., cross-app) scheduler and offered only to a single app at a time. The tight coupling of visibility and allocation makes it infeasible to realize round-by-round auctions where resources need to be visible to many apps but allocated to just one app.

Shared-state fully optimistic schedulers like Omega [99] enforce fully optimistic concurrency control. This means that visibility and allocation go hand-in-hand at the granularity of

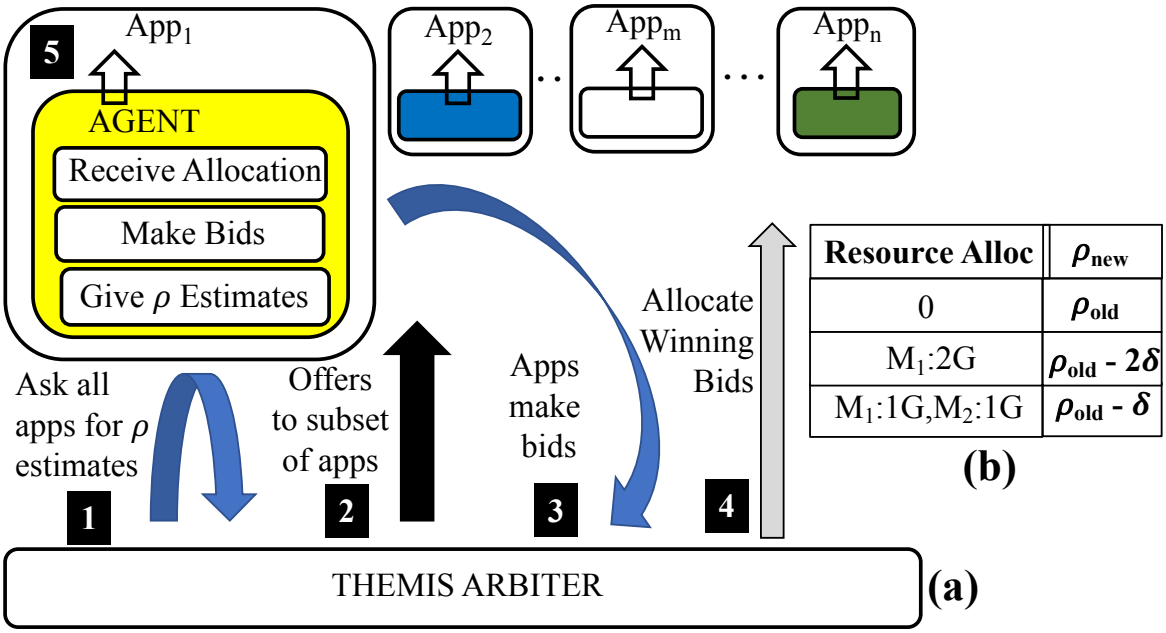


Fig. 3.6 THEMIS Design. (a) Sequence of events in THEMIS - starts with a resource available event and ends with resource allocations. (b) Shows a typical bid valuation table an App submits to ARBITER. Each row has a subset of the complete resource allocation and the improved value of ρ_{new}.

multiple apps. There is full multi-app visibility as all cluster resources and their state is made visible to all apps. Also, all apps contend for resources and resource allocation decisions are made by multiple apps at the same time using transactions. This coupling of visibility and allocation in a lock-free manner makes it hard to realize a global policy like finish-time fairness and also leads to expensive conflict resolution (needed when multiple apps contend for the same resource) when the cluster is highly contented, which is typically the case in shared GPU clusters.

Thus, the properties required by multi-round auctions, i.e., multi-app resource visibility and single-app resource allocation granularity, makes existing architectures ineffective.

3.4.2.2 Two-Level Semi-Optimistic Scheduling

The two-levels in our scheduling architecture comprise of multiple app-schedulers and a cross-app scheduler that we call the ARBITER. The ARBITER has our scheduling logic. The top level per-app schedulers are minimally modified to interact with the ARBITER. Figure 3.6 shows our architecture.

Each GPU in a THEMIS-managed cluster has a lease associated with it. The lease decides the duration of ownership of the GPU for an app. When a lease expires, the resource is made available for allocation. THEMIS's ARBITER pools available resources and runs a round of

the auctions described earlier. During each such round, the resource allocation proceeds in 5 steps spanning 2 phases (shown in Figure 3.6):

The first phase, called the *visibility phase*, spans steps 1–3.

1 The ARBITER asks all apps for current finish-time fair metric estimates. 2 The ARBITER initiates auctions, and makes the same non-binding resource-offer of the available resources to a fraction $f \in [0,1]$ of ML apps with worst finish-time fair metrics (according to round-by-round filtering described earlier). To minimize changes in the ML app scheduler to participate in auctions, THEMIS introduces an AGENT that is co-located with each ML app scheduler. The AGENT serves as an intermediary between the ML app and the ARBITER. 3 The apps examine the resource offer in parallel. Each app's AGENT then replies with a single bid that contains preferences for desired resource allocations.

The second phase, *allocation phase*, spans steps 4–5. 4 The ARBITER, upon receiving all the bids for this round, picks winning bids according to previously described partial allocation algorithm and leftover allocation scheme. It then notifies each AGENT of its winning allocation (if any). 5 The AGENT propagates the allocation to the ML app scheduler, which can then decide the allocation among constituent jobs.

In sum, the two phase resource allocation means that our scheduler enforces *semi-optimistic concurrency control*. Similar to fully optimistic concurrency control, there is multi-app visibility as the cross-app scheduler offers resources to multiple apps concurrently. At the same time, similar to pessimistic concurrency control, the resource allocations are conflict-free guaranteeing exclusive access of a resource to every app.

To enable preparation of bids in step 3, THEMIS implements a narrow API from the ML app scheduler to the AGENT that enables propagation of app-specific information. An AGENT's bid contains a *valuation function* ($\rho(.)$) that provides, for each resource subset, an estimate of the finish-time fair metric the app will achieve with the allocation of the resource subset. We describe how this is calculated next.

3.4.3 AGENT and AppScheduler Interaction

An AGENT co-resides with an app to aid participation in auctions. We now describe how AGENTs prepare bids based on inputs provided by apps, the API between an AGENT and its app, and how AGENTs integrate with current hyper-parameter optimization schedulers.

3.4.3.1 Single-Job ML Apps

For ease of explanation, we first start with the simple case of an ML app that has just one ML training job which can use at most job_demand_{max} GPUs. We first look at calculation of the

finish-time fair metric, ρ. We then look at a multi-job app example so as to better understand the various steps and interfaces in our system involved in a multi-round auction.

Calculating $\rho(\overrightarrow{G})$. Equation 3.1 shows the steps for calculating ρ for a single job given a GPU allocation of $\overrightarrow{G}$ in a cluster C with R_C GPUs. When calculating ρ we assume that the allocation $\overrightarrow{G}$ is binding till job completion.

$$\begin{aligned}
\rho(\overrightarrow{G}) &= T_{sh}(\overrightarrow{G})/T_{id} \\
T_{sh} &= T_{current} - T_{start} + \\
&\quad iter_left * iter_time(\overrightarrow{G}) \\
T_{id} &= T_{cluster} * N_{avg} \\
iter_time(\overrightarrow{G}) &= \frac{iter_time_serial * \mathscr{S}(\overrightarrow{G})}{min(||\overrightarrow{G}||_1, job_demand_{max})} \\
T_{cluster} &= \frac{iter_total * iter_serial_time}{min(R_C, job_demand_{max})}
\end{aligned} \tag{3.1}$$

T_{sh} is the shared finish-time and is a function of the allocation $\overrightarrow{G}$ that the job receives. For the single job case, it has two terms. First, is the time elapsed ($= T_{current} - T_{start}$). Time elapsed also captures any queuing delays or starvation time. Second, is the time to execute remaining iterations which is the product of the number of iterations left ($iter\_left$) and the iteration time ($iter\_time(\overrightarrow{G})$). $iter\_time(\overrightarrow{G})$ depends on the allocation received. Here, we consider the common-case of the ML training job executing synchronous SGD. In synchronous SGD, the work in an iteration can be parallelized across multiple workers. Assuming linear speedup, this means that the iteration time is the serial iteration time ($iter\_time\_serial$) reduced by a factor of the number of GPUs in the allocation, $||\overrightarrow{G}||_1$ or job_demand_{max} whichever is lesser. However, the linear speedup assumption is not true in the common case as network overheads are involved. We capture this via a slowdown penalty, $\mathscr{S}(\overrightarrow{G})$, which depends on the placement of the GPUs in the allocation. Values for $\mathscr{S}(\overrightarrow{G})$ can typically be obtained by profiling the job offline for a few iterations. [3] The slowdown is captured as a multiplicative factor, $\mathscr{S}(\overrightarrow{G}) \geq 1$, by which T_{sh} is increased.

T_{id} is the estimated finish-time in an independent $\frac{1}{N_{avg}}$ cluster. N_{avg} is the average contention in the cluster and is the weighted average of the number of apps in the system during the lifetime of the app. We approximate this as the finish-time of the app in the whole cluster, $T_{cluster}$ multiplied by the average contention. $T_{cluster}$ assumes linear speedup when

[3] $\mathscr{S}(\overrightarrow{G})$ can also be calculated in an online fashion. First, we use crude placement preference estimates to begin with for single machine (=1), cross-machine (=1.1), cross-rack (=1.3) placement. These are replaced with accurate estimates by profiling iteration times when the ARBITER allocates unseen placements. The multi-round nature of allocations means that errors in early estimates do not have a significant effect.

the app executes with all the cluster resources R_C or maximum app demand whichever is lesser. It also assumes no slowdown. Thus, it is approximated as $\frac{iter\_total * iter\_serial\_time}{min(R_C, job\_demand_{max})}$.

3.4.3.2 Generalizing to Multiple-Job ML Apps

ML app schedulers for hyper-parameter optimization systems typically go from aggressive exploration of hyper-parameters to aggressive exploitation of best hyper-parameters. While there are a number of different algorithms for choosing the best hyper-parameters [77, 28] to run, we focus on early stopping criteria as this affects the finish time of ML apps.

As described in prior work [51], automatic stopping algorithms can be divided into two categories: Successive Halving and Performance Curve Stopping. We next discuss how to compute T_{sh} for each case.

Successive Halving refers to schemes which start with a total time or iteration budget B and apportion that budget by periodically stopping jobs that are not promising. For example, if we start with n hyper parameter options, then each one is submitted as a job with a demand of 1 GPU for a fixed number of iterations I. After I iterations, only the best $\frac{n}{2}$ ML training jobs are retained and assigned a maximum demand of 2 GPUs for the same number of iterations I. This continues until we are left with 1 job with a maximum demand of n GPUs. Thus there are a total of $log_2 n$ *phases* in Successive Halving. This scheme is used in Hyperband [77] and Google Vizier [51].

We next describe how to compute T_{sh} and T_{id} for successive halving. We assume that the given allocation $\overrightarrow{G}$ lasts till app completion and the total time can be computed by adding up the time the app spends for each phase. Consider the case of phase i which has $J = \frac{n}{2^{i-1}}$ jobs. Equation 3.2 shows the calculation of $T_{sh(i)}$, the shared finish time of the phase. We assume a separation of concerns where the hyper-parameter optimizer can determine the optimal allocation of GPUs *within a phase* and thus estimate the value of $\mathscr{S}(\overrightarrow{G_j})$. Along with $iter\_left$, $serial\_iter\_time$, the AGENT can now estimate $T_{sh(j)}$ for each job in the phase. We mark the phase as finished when the slowest or last job in the app finishes the phase (max_j). Then the shared finish time for the app is the sum of the finish times of all constituent phases.

To estimate the ideal finish-time we compute the total time to execute the app on the full cluster. We estimate this using the budget B which represents the aggregate work to be done and, as before, we assume linear speedup to the maximum number of GPUs the app can use app_demand_{max}.

$$
\begin{aligned}
T_{sh(i)} &= max_j\{T(\overrightarrow{G_j})\} \\
T_{sh} &= \sum_i T_{sh(i)} \\
T_{cluster} &= \frac{B}{min(R_C, app\_demand_{max})} \\
T_{id} &= T_{cluster} * N_{avg}
\end{aligned}
\tag{3.2}
$$

The AGENT generates ρ using the above procedure for all possible subsets of $\{\overrightarrow{G}\}$ and produces a bid table similar to the one shown in Table 3.2 before. The API between the AGENT and hyper-parameter optimizer is shown in Figure 3.7 and captures the functions that need to implemented by the hyper-parameter optimizer.

Performance Curve Stopping refers to schemes where the convergence curve of a job is extrapolated to determine which jobs are more promising. This scheme is used by Hyperdrive [98] and Google Vizier [51]. Computing T_{sh} proceeds by calculating the finish time for each job that is currently running by estimating the iteration at which the job will be terminated (thus T_{sh} is determined by the job that finishes last). As before, we assume that the given allocation $\overrightarrow{G}$ lasts till app completion. Since the estimations are usually probabilistic, i.e., the iterations at which the job will converge has an error bar, we over-estimate and use the most optimistic convergence curve that results in the maximum forecasted completion time for that job. As the job progresses, the estimates of the convergence curve get more accurate and improves the accuracy of the estimated finish time T_{sh}. The API implemented by the hyper-parameter optimizer is simpler and only involves getting a list of running jobs as shown in Figure 3.7.

We next present an end-to-end example of a multi-job app showing our mechanism in action.

3.4.3.3 End-to-end Example.

We now run through a simple example that exercises the various aspects of our API and the interfaces involved.

Consider a 16 GPU cluster and an ML app that has 4 ML jobs and uses successive halving, running along with 3 other ML apps in the same cluster. Each job in the app is tuning a different hyper-parameter and the serial time taken per iteration for the jobs are $80, 100, 100, 120$ seconds respectively.[4] The total budget for the app is $10,000$ seconds of GPU time and we assume the job_demand_{max} is 8 GPUs and $\mathscr{S}(\overrightarrow{G}) = 1$.

[4]The time per iteration depends on the nature of the hyper-parameter being tuned. Some hyper-parameters like batch size or quantization used affect the iteration time while others like learning rate don't.

```
class JobInfo(int itersRemaining,
              float avgTimePerIter,
              float localitySensitivity);
// Successive Halving
List<JobInfo> getJobsInPhase(int phase,
                             List<Int> gpuAlloc);
int getNumPhases();
// Performance Curve
List<JobInfo> getJobsRemaining(List<Int> gpuAlloc);
```

Fig. 3.7 API between AGENT and hyperparameter optimizer

Given we start with 4 ML jobs, the hyper-parameter optimizer divides this into 3 phases each having $4, 2, 1$ jobs, respectively. To evenly divide the budget across the phases, the hyper-parameter optimizer budgets $\approx 8, 16, 36$ iterations in each phase. First we calculate the T_{id} by considering the budget, total cluster size, and cluster contention as: $\frac{10000 \times 4}{16} = 2500$s.

Next, we consider the computation of T_{sh} assuming that 16 GPUs are offered by the ARBITER. The AGENT now computes the bid for each subset of GPUs offered. Consider the case with 2 GPUs. In this case in the first phase we have 4 jobs which are serialized to run 2 at a time. This leads to $T_{sh(1)} = (120 \times 8) + (80 \times 8) = 1600$ seconds. (Assume two 100s jobs run serially on one GPU, and the 80 and 120s jobs run serially on the other. T_{sh} is the time when the last job finishes.)

When we consider the next stage the hyper-parameter optimizer currently does not know which jobs will be chosen for termination. We use the *median* job (in terms of per-iteration time) to estimate $T_{sh(i)}$ for future phases. Thus, in the second phase we have 2 jobs so we run one job on each GPU each of which we assume to take the median 100 seconds per iteration leading to $T_{sh(2)} = (100 \times 16) = 1600$ seconds. Finally for the last phase we have 1 job that uses 2 GPUs and runs for 36 iterations leading to $T_{sh(3)} = \frac{(100 \times 36)}{2} = 1800$ (again, the "median" jobs takes 100s per iteration). Thus $T_{sh} = 1600 + 1600 + 1800 = 5000$ seconds, making $\rho = \frac{5000}{2500} = 2$. Note that since placement did not matter here we considered any 2 GPUs being used. Similarly ignoring placement, the bids for the other allocations are shown in Table 3.3.

We highlight a few more points about our example above. If the jobs that are chosen for the next phase do not match the median iteration time then the estimates are revised in the next round of the auction. For example, if the jobs that are chosen for the next round have iteration time $120, 100$ then the above bid will be updated with $T_{sh(2)} = (120 \times 16) = 3200$[5] and $T_{sh(3)} = \frac{(120 \times 36)}{2} = 2160$. Further, we also see that the $job\_demand_{max} = 8$ means that the ρ value for 16 GPUs does not linearly decrease from that of 8 GPUs.

[5]Because the two jobs run on one GPU each, and the 120s-per-iteration job is the last to finish in the phase

$\|\|\overrightarrow{G}\|\|_1$	0	1	2	4	8	16
ρ	ρ_{old}	4	2	1	0.5	0.34

Table 3.3 Example of bids submitted by AGENT

3.5 Implementation

We implement THEMIS on top of a recent release of Apache Hadoop YARN [109] (version 3.2.0) which includes, Submarine [26], a new framework for running ML training jobs atop YARN. We modify the Submarine client to support submitting a group of ML training jobs as required by hyper-parameter exploration apps. Once an app is submitted, it is managed by a Submarine Application Master (AM) and we make changes to the Submarine AM to implement the ML app scheduler (we use Hyperband [77]) and our AGENT.

To prepare accurate bids, we implement a profiler in the AM that parses TensorFlow logs, and tracks iteration times and loss values for all the jobs in an app. The allocation of a job changes over time and iteration times are used to accurately estimate the placement preference ($\mathscr{S}$) for different GPU placements. Loss values are used in our Hyperband implementation to determine early stopping. THEMIS's ARBITER is implemented as a separate module in YARN RM. We add gRPC-based interfaces between the AGENT and the ARBITER to enable offers, bids, and final winning allocations. Further, the ARBITER tracks GPU leases to offer reclaimed GPUs as a part of the offers.

All the jobs we use in our evaluation are TensorFlow programs with configurable hyper-parameters. To handle allocation changes at runtime, the programs checkpoint model parameters to HDFS every few iterations. After a change in allocation, they resume from the most recent checkpoint.

3.6 Evaluation

We evaluate THEMIS on a 64 GPU cluster and also use a event-driven simulator to model a larger 256 GPU cluster. We compare against other state-of-the-art ML schedulers. Our evaluation shows the following key highlights -

- THEMIS is better than other schemes on finish-time fairness while also offering better cluster efficiency (Figure 3.9-3.10-3.11-3.12).

- THEMIS's benefits compared to other schemes improve with increasing fraction of placement sensitive apps and increasing contention in the cluster, and these improvements hold even with errors – random and strategic – in finish-time fair metric estimations (Figure 3.14-3.18).

• THEMIS enables a trade-off between finish-time fairness in the long-term and placement efficiency in the short-term. Sensitivity analysis (Figure 3.19) using simulations show that $f = 0.8$ and a lease time of 10 minutes gives maximum fairness while also utilizing the cluster efficiently.

3.6.1 Experimental Setup

Testbed Setup. Our testbed is a 64 GPU, 20 machine cluster on Microsoft Azure [11]. We use NC-series instances. We have 8 NC12-series instances each with 2 Tesla K80 GPUs and 12 NC24-series instances each with 4 Tesla K80 GPUs.

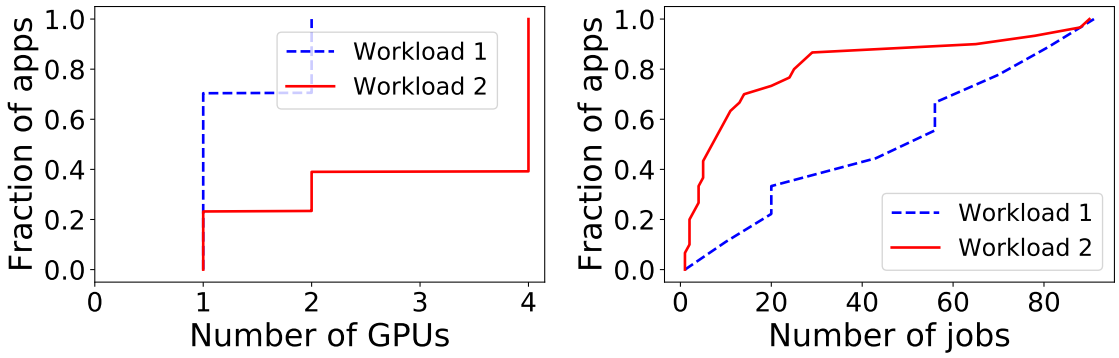
Simulator. We develop an event-based simulator to evaluate THEMIS at large scale. The simulator assumes that estimates of the loss function curves for jobs are known ahead of time so as to predict the total number of iterations for the job. Unless stated otherwise, all simulations are done on a heterogeneous 256 GPU cluster. Our simulator assumes a 4-level hierarchical locality model for GPU placements. Individual GPUs fit onto *slots* on *machines* occupying different cluster *racks*.[6]

Workload. We experiment with 2 different traces that have different workload characteristics in both the simulator and the testbed - **(i) Workload 1.** A publicly available trace of DNN training workloads at Microsoft [70, 85]. We scale-down the trace, using a two week snapshot and focus on subset of jobs from the trace that correspond to hyper-parameter exploration jobs triggered by Hyperdrive. **(ii) Workload 2.** We use the app arrival times from Workload 1, generate jobs per app using the successive halving pattern characteristic of the Hyperband algorithm [77], and increase the number of tasks per job compared to Workload 1. The distribution of number of tasks per job and number of jobs per app for the two workloads is shown in Figure 3.8.

The traces comprise of models from three categories - computer vision (CV - 10%), natural language processing (NLP - 60%) and speech (Speech - 30%). We use the same mix of models for each category as outlined in Gandiva [117]. We summarize the models in Table 3.4.

Baselines. We compare THEMIS against four state-of-the-art ML schedulers - Gandiva [117], Tiresias [56], Optimus [93], SLAQ [126]; these represent the best possible baselines for maximizing efficiency, fairness, aggregate throughput, and aggregate model quality, respectively. We also compare against two scheduling disciplines - shortest remaining time first (SRTF) and shortest remaining service first (SRSF) [56]; these represent baselines for minimizing average job completion time (JCT) with efficiency as secondary concern and minimizing

[6]The heterogeneous cluster consists of 16 8-GPU machines (4 slots and 2 GPUs per slot), 6 4-GPU machines (4 slots and 1 GPU per slot), and 16 1-GPU machines



(a) CDF GPUs per job

(b) CDF jobs per app

Fig. 3.8 Details of 2 workloads used for evaluation of THEMIS

	Model	Type	Dataset
10%	Inception-v3 [105]	CV	ImageNet [41]
	AlexNet [75]	CV	ImageNet
	ResNet50 [61]	CV	ImageNet
	VGG16 [103]	CV	ImageNet
	VGG19 [103]	CV	ImageNet
60%	Bi-Att-Flow [100]	NLP	SQuAD [97]
	LangModel [125]	NLP	PTB [82]
	GNMT [116]	NLP	WMT16 [114]
	Transformer [108]	NLP	WMT16
30%	WaveNet [91]	Speech	VCTK [118]
	DeepSpeech [58]	Speech	CommonVoice [39]

Table 3.4 Models used in our trace.

average JCT with fairness as secondary concern, respectively. We implement these baselines in our testbed as well as the simulator as described below:

Ideal Efficiency Baseline - Gandiva. Gandiva improves cluster utilization by packing jobs on as few machines as possible. In our implementation, Gandiva introspectively profiles ML job execution to infer placement preferences and migrates jobs to better meet these placement preferences. On any resource availability, all apps report their placement preferences and we allocate resources in a greedy highest preference first manner which has the effect of maximizing the average placement preference across apps. We do not model time-slicing and packing of GPUs as these system-level techniques can be integrated with THEMIS as well and would benefit Gandiva and THEMIS to equal extents.

Ideal Fairness Baseline - Tiresias. Tiresias defines a new service metric for ML jobs – the aggregate GPU-time allocated to each job – and allocates resources using the Least Attained Service (LAS) policy so that all jobs obtain equal service over time. In our implementation,

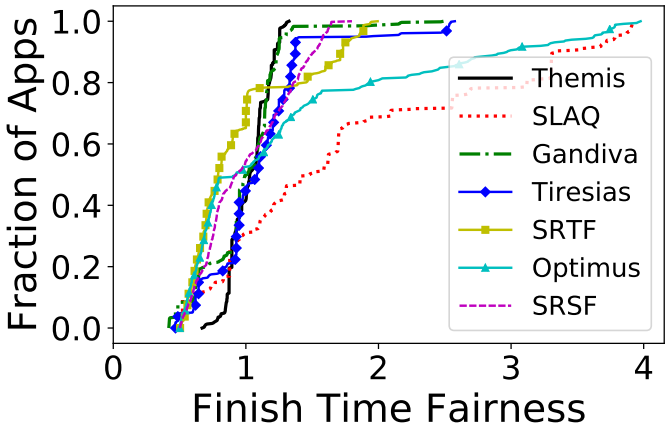


Fig. 3.9 [TESTBED] Comparison of finish-time fairness across schedulers with Workload 1

on any resource availability, all apps report their service metric and we allocate the resource to apps that have the least GPU service.

Ideal Aggregate Throughput Baseline - Optimus. Optimus proposes a throughput scaling metric for ML jobs – the ratio of new job throughput to old job throughput with and without an additional GPU allocation. On any resource availability, all apps report their throughput scaling and we allocate resources in order of highest throughput scaling metric first.

Ideal Aggregate Model Quality - SLAQ. SLAQ proposes a greedy scheme for improving aggregate model quality across all jobs. In our implementation, on any resource availability event, all apps report the decrease in loss value with allocations from the available resources and we allocate these resources in a greedy highest loss first manner.

Ideal Average App Completion Time - SRTF, SRSF. For SRTF, on any resource availability, all apps report their remaining time with allocations from the available resource and we allocate these resources using SRTF policy. Efficiency is a secondary concern with SRTF as better packing of GPUs leads to shorter remaining times.

SRSF is a service-based metric and approximates gittins index policy from Tiresias. In our implementation, we assume accurate knowledge of remaining service and all apps report their remaining service and we allocate one GPU at a time using SRSF policy. Fairness is a secondary concern as shorter service apps are preferred first as longer apps are more amenable to make up for lost progress due to short-term unfair allocations.

Metrics. We use a variety of metrics to evaluate THEMIS.

(i) Finish-time fairness: We evaluate the fairness of schemes by looking at the finish-time fair metric (ρ) distribution and the maximum value across apps. A tighter distribution and a lower value of maximum value of ρ across apps indicate higher fairness. **(ii) GPU Time:** We use *GPU Time* as a measure of how efficiently the cluster is utilized. For two

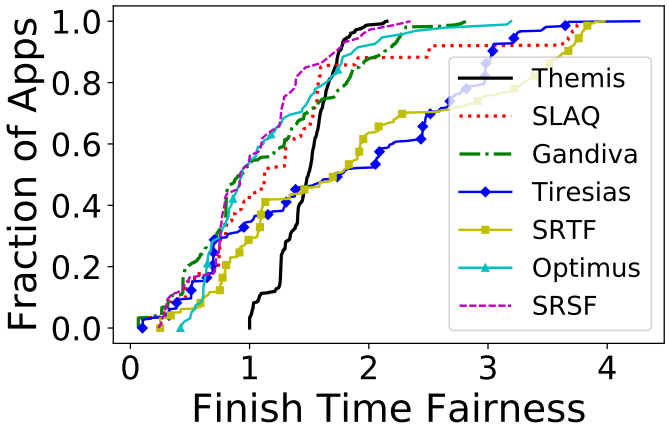


Fig. 3.10 [TESTBED] Comparison of finish-time fairness across schedulers with Workload 2

scheduling schemes S_1 and S_2 that have GPU times G_1 and G_2 for executing the same amount of work, S_1 utilizes the cluster more efficiently than S_2 if $G_1 < G_2$. **(iii) Placement Score:** We give each allocation a placement score (≤ 1). This is inversely proportional to slowdown, $\mathscr{S}$, that app experiences due to this allocation. The slowdown is dependent on the ML app properties and the network interconnects between the allocated GPUs. A placement score of 1.0 is desirable for as many apps as possible.

3.6.2 Macrobenchmarks

In our testbed, we evaluate THEMIS against all baselines on all the workloads. We set the fairness knob value f as 0.8 and lease as 10 minutes, which is informed by our sensitivity analysis results in Section 3.6.4.

Figure 3.9-3.10 shows the distribution of finish-time fairness metric, ρ, across apps for THEMIS and all the baselines. We see that THEMIS has a narrower distribution for the ρ values which means that THEMIS comes closest to giving all jobs an equal sharing incentive. Also, THEMIS gives 2.2X to 3.25X better (smaller) maximum ρ values compared to all baselines.

Figure 3.11-3.12 shows a comparison of the efficiency in terms of the aggregate GPU time to execute the complete workload. Workload 1 has similar efficiency across THEMIS and the baselines as all jobs are either 1 or 2 GPU jobs and almost all allocations, irrespective of the scheme, end up as efficient. With workload 2, THEMIS betters Gandiva by ~4.8% and outperforms SLAQ by ~250%. THEMIS is better because global visibility of app placement preferences due to the auction abstraction enables globally optimal decisions. Gandiva in contrast takes greedy locally optimal packing decisions.

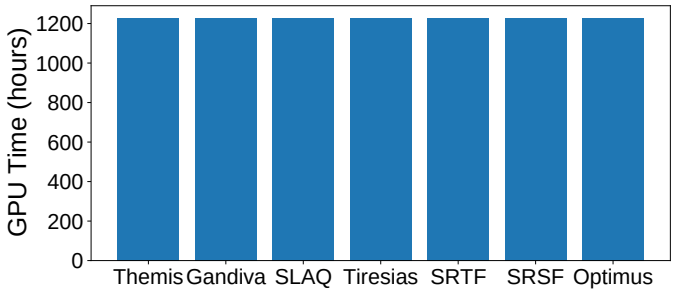


Fig. 3.11 [TESTBED] Comparison of total GPU times across schemes with Workload 1. Lower GPU time indicates better utilization of the GPU cluster

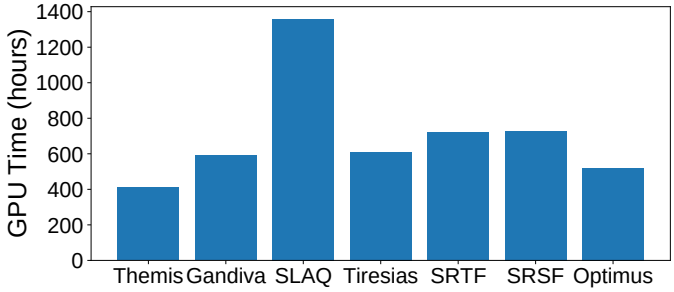


Fig. 3.12 [TESTBED] Comparison of total GPU times across schemes with Workload 2. Lower GPU time indicates better utilization of the GPU cluster

3.6.2.1 Sources of Improvement

In this section, we deep-dive into the reasons behind the wins in fairness and cluster efficiency in THEMIS.

Table 3.5 compares the finish-time fair metric value for a pair of short- and long-lived apps from our testbed run for THEMIS and Tiresias. THEMIS offers better sharing incentive for both the short and long apps. THEMIS induces altruistic behavior in long apps. We attribute this to our choice of ρ metric. With less than ideal allocations, even though long apps see an increase in T_{sh}, their ρ values do not increase drastically because of a higher T_{id} value in the denominator. Whereas, shorter apps see a much more drastic degradation, and our round-by-round filtering of farthest-from-finish-time fairness apps causes shorter apps to participate in auctions more often. Tiresias offers poor sharing incentive for short apps as it treats short- and long-apps as the same. This only worsens the sharing incentive for short apps.

Figure 3.13 shows the distribution of placement scores for all the schedulers. THEMIS gives the best placement scores (closer to 1.0 is better) in workload 2, with Gandiva and Optimus coming closest. Workload 1 has jobs with very low GPU demand and almost all allocations have a placement score of 1 irrespective of the scheme. Other schemes are poor

Job Type	GPU Time	# GPUs	ρ_{THEMIS}	$\rho_{Tiresias}$
Long Job	~580 mins	4	~1	~0.9
Short Job	~83 mins	2	~1.2	~1.9

Table 3.5 [TESTBED] Details of 2 jobs to understand the benefits of THEMIS

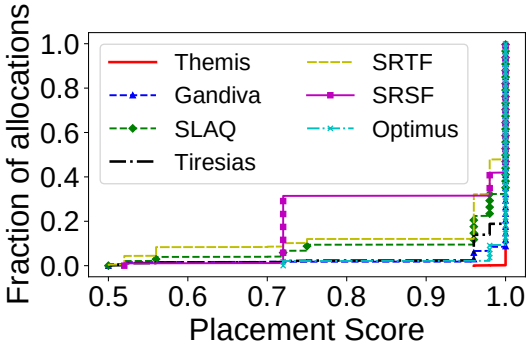


Fig. 3.13 [TESTBED] CDF of placement scores across schemes

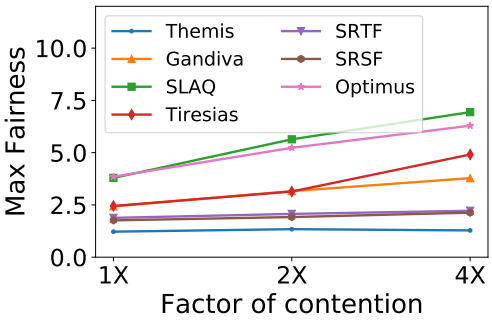


Fig. 3.14 [TESTBED] Impact of contention on finish-time fairness

as they do not account for placement preferences. Gandiva does greedy local packing and Optimus does greedy throughput scaling and are not as efficient because they are not globally optimal.

3.6.2.2 Effect of Contention

In this section, we analyze the effect of contention on finish-time fairness. We decrease the size of the cluster to half and quarter the original size to induce a contention of $2X$ and $4X$ respectively. Figure 3.14 shows the change in max value of ρ as the contention changes with workload 1. THEMIS is the only scheme that maintains *sharing incentive* even in high contention scenarios. SRSF comes close as it preferably allocates resources to shorter service apps. This behavior is similar to that in THEMIS. THEMIS induces altruistic shedding of resources by longer apps (Section 3.6.2.1), giving shorter apps a preference in allocations during higher contention.

3.6.2.3 Systems Overheads

From our profiling of the experiments above, we find that each AGENT spends 29 (334) milliseconds to compute bids at the median (95-%). The 95 percentile is high because enumeration of possible bids needs to traverse a larger search space when the number of resources up for auction is high.

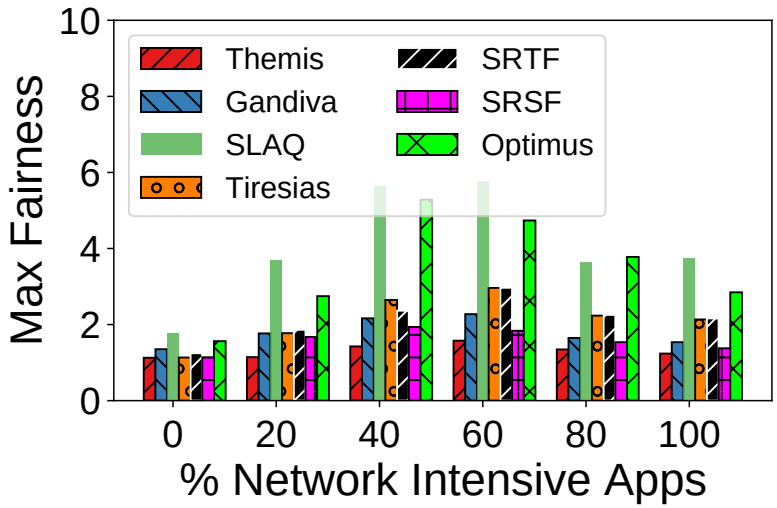


Fig. 3.15 [SIMULATOR] Impact of placement preferences for varying mix of compute- and network-intensive apps on max ρ

The ARBITER uses Gurobi [57] to compute partial allocation of resources to apps based on bids. This computation takes 354 (1398) milliseconds at the median (95-%ile). The high tail is once again observed when both the number of offered resources and the number of apps bidding are high. However, the time is small relative to lease time. The network overhead for communication between the ARBITER and individual apps is negligible since we use the existing mechanisms used by Apache YARN.

Upon receiving new resource allocations, the AGENT changes (adds/removes) the number of GPU containers available to its app. This change takes about 35 (50) seconds at the median (95-%ile), i.e., an overhead of 0.2% (2%) of the app duration at the median (95-%ile). Prior to relinquishing control over its resources, each application must checkpoint its set of parameters. We find that that this is model dependent but takes about 5-10 seconds on an average and is driven largely by the overhead of check-pointing to HDFS.

3.6.3 Microbenchmarks

Placement Preferences: We analyze the impact on finish-time fairness and cluster efficiency as the fraction of network-intensive apps in our workload increases. We synthetically construct 6 workloads and vary the percentage of network-intensive apps in these workloads from 0%-100%.

From Figure 3.15, we notice that *sharing incentive* degrades most when there is a heterogeneous mix of compute and network intensive apps (at 40% and 60%). THEMIS has a max ρ value closest to 1 across all scenarios and is the only scheme to ensure sharing

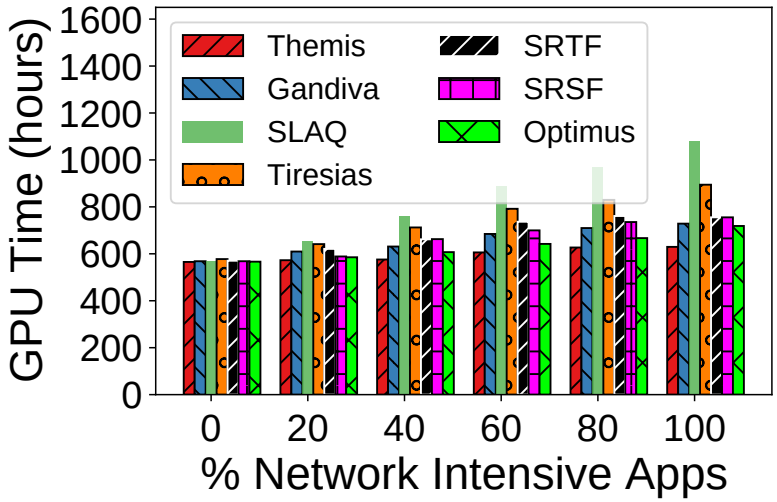


Fig. 3.16 [SIMULATOR] Impact of placement preferences for varying mix of compute- and network-intensive apps on GPU Time

incentive. When the workload consists solely of network-intensive apps, THEMIS performs ~1.24 to 1.77X better than existing baselines on max fairness.

Figure 3.16 captures the impact on cluster efficiency. With only compute-intensive apps, all scheduling schemes utilize the cluster equally efficiently. As the percentage of network intensive apps increases, THEMIS has lower GPU times to execute the same workload. This means that THEMIS utilizes the cluster more efficiently than other schemes. In the workload with 100% network-intensive apps, THEMIS performs ~8.1% better than Gandiva (state-of-the-art for cluster efficiency).

Error Analysis: Here, we evaluate the ability of THEMIS to handle errors in estimation of number of iterations and the slowdown ($\mathscr{S}$). For this experiment, we assume that all apps are equally susceptible to making errors in estimation. The percentage error is sampled at random from [-X, X] range for each app. Figure 3.17 shows the changes in max finish-time fairness as we vary X. Even with $X = 20\%$, the change in max finish-time fairness is just 10.76% and is not significant.

Truth-Telling: To evaluate strategy-proofness, we use simulations. We use a cluster of 64 GPUs with 8 identical apps with equivalent placement preferences. The cluster has a single 8 GPU machine and the others are all 2 GPU machines. The most preferred allocation in this cluster is the 8 GPU machine. We assume that there is a single strategically lying app and 7 truthful apps. In every round of auction it participates in, the lying app over-reports the slowdown with staggered machine placement or under-reports the slowdown with dense machine placement by $X\%$. Such a strategy would ensure higher likelihood of winning the 8 GPU machine. We vary the value of X in the range $[0, 100]$ and analyze the lying app's

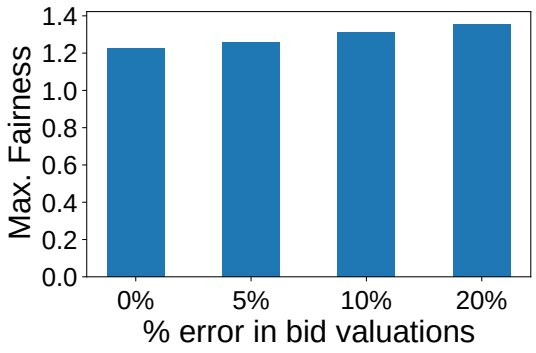


Fig. 3.17 [SIMULATOR] Impact of error in bid values on max fairness

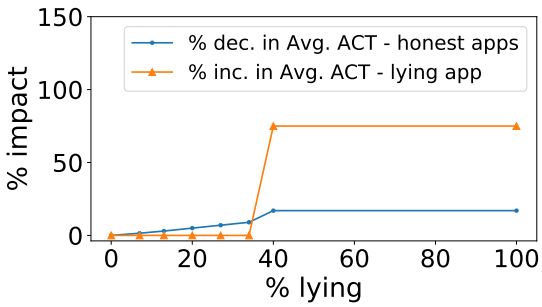


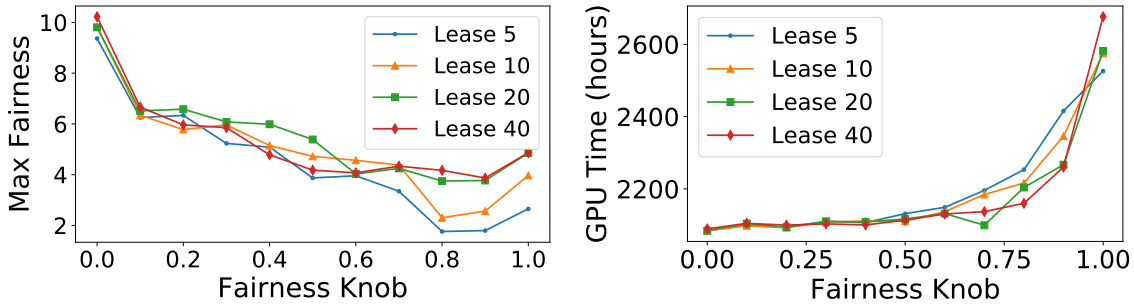
Fig. 3.18 [SIMULATOR] Strategic lying is detrimental

completion time and the average app completion time of the truthful apps in Figure 3.18. We see that at first the lying app does not experience any decrease in its own app completion time. On the other hand, we see that the truthful apps do better on their average app completion time. This is because the hidden payment from the partial allocation mechanism in each round of the auction for the lying app remains the same while the payment from the rest of the apps keeps decreasing. We also observe that there is a sudden tipping point at $X > 34\%$. At this point, there is a sudden increase in the hidden payment for the lying app and it loses a big chunk of resources to other apps. In essence, THEMIS incentivizes truth-telling.

3.6.4 Sensitivity Analysis

We use simulations to study THEMIS's sensitivity to fairness knob f and the lease time. Figure 3.19 (a) shows the impact on max ρ as we vary the fairness knob f. We observe that filtering $(1-f)$ fraction of apps helps with ensuring better *sharing incentive*. As f increases from 0 to 0.8, we observe that fairness improves. Beyond $f = 0.8$, max fairness worsens by around a factor of $1.5X$. We see that the quality of sharing incentive, captured by max ρ, degrades at $f = 1$ because we observe that only a single app with highest ρ value participates in the auction. This app is forced sub-optimal allocations because of poor placement of available resources with respect to the already allocated resources in this app. We also observe that smaller lease times promote better fairness since frequently filtering apps reduces the time that queued apps wait for an allocation.

Figure 3.19 (b) shows the impact on the efficiency of cluster usage as we vary the fairness knob f. We observe that the efficiency decreases as the value of f increases. This is because the number of apps that can bid for an offer reduces as we increase f leading to fewer opportunities for the ARBITER to pack jobs efficiently. Lower lease values mean than models



(a) Impact on Max Fairness

(b) Impact on GPU Time

Fig. 3.19 [SIMULATOR] Sensitivity of fairness knob and lease time.

need to be check-pointed more often (GPUs are released on lease expiry) and hence higher lease values are more efficient.

Thus we choose $f = 0.8$ and $lease = 10$ minutes.

3.7 Related Work

Cluster scheduling for ML workloads has been targeted by a number of recent works including SLAQ [126], Gandiva [117], Tiresias [56] and Optimus [93]. These systems target different objectives and we compare against them in Section 3.6.

We build on rich literature on cluster scheduling disciplines [48, 55, 54, 53] and two level schedulers [63, 110, 99]. While those disciplines/schedulers don't apply to our problem, we build upon some of their ideas, e.g., resource offers in [63]. Sharing incentive was outlined by DRF [48], but we focus on long term fairness with our finish-time metric. Tetris [53] proposes resource-aware packing with an option to trade-off for fairness using multi-dimensional bin-packing as the mechanism for achieving that. In THEMIS, we instead focus on fairness with an option to trade-off for placement-aware packing, and use auctions as our mechanism.

Some earlier schemes [54, 55] also attempted to emulate the long term effects of fair allocation. Around occasional barriers, unused resources are re-allocated across jobs. THEMIS differs in many respects: First, earlier systems focus on batch analytics. Second, earlier schemes rely on instantaneous resource-fairness (akin to DRF), which has issues with placement-preference unawareness and not accounting for long tasks. Third, in the ML context there are no occasional barriers. While barriers do arise due to synchronization of parameters in ML jobs, they happen at *every* iteration. Also, resources unilaterally given up by a job may not be usable by another job due to placement preferences.

Chapter 4

Better Together: Jointly Optimizing ML Collective Scheduling and Execution Planning using SYNDICATE

4.1 Introduction

Training machine learning (ML) models is a common-case workload at any data-driven enterprise. To keep up with evolving data and maintain a competitive edge, enterprises are employing more sophisticated features and more complex model architectures, and attempting to train faster at ever larger scales and to deploy high-quality models frequently.

These trends are exemplified by the deep learning recommendation model (DLRM). DLRM is used in recommendation systems at several large organizations. These models use a mixture of continuous and categorical features obtained from user data. The model architectures, which are themselves rapidly evolving, uses a mixture of multi-layer perceptrons and embedding table lookups. The model capacity and compute is increasing exponentially year-on-year [86].

At production scale, state-of-the-art models such as DLRM use a mixture of data and model parallelism to efficiently scale-out to a large number of machines in the training cluster. This induces rich communication collectives such as all-reduce, all-to-all, collective-permute, and all-gather [76, 86]. The resulting communication operations (comm-ops) are a major bottleneck to end-to-end training performance [86].

Evolution in networking infrastructure in training clusters [104, 119] (to include faster interconnects such as NVLink/NVSwitch, RoCE, Infiniband and support faster transports such as Remote Direct Memory Access (RDMA)) does not in itself help address these bottle-

necks. These advancements need to be coupled with effective computation-communication scheduling and execution planning optimizations. These optimizations hide communication by maximizing overlap with compute and help maximize utilization of the networking infrastructure.

Unfortunately, existing scheduling optimizations [59, 94, 69] and execution planners [34, 106, 37, 113, 72] fall short. These works make several restrictive assumptions limiting their applicability to simplistic models, training settings, and networks. Communication schedulers make assumptions about the model and training architecture (simple layer-by-layer models [94] with data-parallel training) or deployment mode (Parameter Server-based [69, 59]), and the execution planners make simplifying assumptions about the nature of comm-ops (only all-reduce [34, 106, 37, 113, 72] or only push-pull [72]).

Moreover, existing solutions are point solutions in the optimization space and fail to *jointly* optimize scheduling and execution planning concerns. Schedulers today are unaware of the optionality during execution planning, such as parallel execution of two comm-ops over non-overlapping network communication channels, and might impose orders that fail to leverage such opportunities during execution planning. As a result, they leave significant room for optimization.

We seek a comm-op optimization framework that jointly optimizes planning and scheduling, applies to state-of-the-art large models with complex communication patterns, is generalizable to future large models and arbitrary network interconnects. Our framework should also encapsulates all possible optimization axes, and enables a systematic, thorough, automatic search through the space for optimal strategies.

Enabling systematic joint optimization of scheduling and execution planning is challenging. First, the communication systems stacks used for ML training today place scheduling and execution planning concerns in two different layers. The scheduler is co-located with ML training frameworks (such as PyTorch, TensorFlow) and the execution planner is co-located with communication libraries (such as NCCL, MPI). These are governed by two different developer communities and the scheduler interacts with the execution planner via a narrow, one-way API to just submit comm-ops. Moreover, the scheduler and the execution planner only accommodate fast, deterministic procedures so as to enable tight co-ordination across worker processes that peer with each other using parallel programming frameworks (such as MPI) during training. Second, scheduling happens at the very coarse granularity of collectives submitted by the training application which limits scheduling flexibility as it leads to fewer opportunities to reorder communication work in time and efficiently pack communication work in space, i.e., over the heterogeneous mix of communication channels and bandwidth available in the networking infrastructure.

In this work, we propose SYNDICATE, a system for joint optimization of scheduling and execution planning concerns. We make the following key contributions:

- SYNDICATE proposes a novel abstraction, the motif, to break large communication work in comm-ops into smaller units of communication work. Motifs afford greater flexibility, by helping pack and order communication work so as to maximize network multipath utilization and to maximize overlap with compute.
- Similar to query optimization backed by a well-defined relational algebra, we present a novel algebra atop motifs that systematically codifies the search space of correct, composable motif operators used to transform comm-ops into motifs and enables comm-op optimization.
- SYNDICATE rethinks the interfaces in the commmunication stack and enables joint optimization via the joint action of a control plane and a data plane. The former executes a time-intensive, non-deterministic joint optimization out-of-band without blocking the latter which enables fast execution of tightly co-ordinated motifs.
- We blend techniques used for optimal tensor operator fragmentation [71], DAG scheduling [55] and query replanning [80] to probabilistically search the joint optimization space to yield near-optimal comm-op optimization plans. We also introduces a novel shim-layer above existing communication libraries to enforce these plans.

We implement the enforcer as a shim-layer atop existing communication libraries by extending the `torch-ucc` interface; the joint optimizer as a separate python process; and enable safe interaction between the central optimizer and the enforcer via a two phase commit protocol. We present the evaluation of real-world DLRM models used in production at our organization on a 128-GPU cluster with rich multipath opportunities. SYNDICATE demonstrates 38–74% faster training than the closest state-of-the-art [94] and is better than hand-optimized trainers.

4.2 Background

Model training deployments are constantly evolving. The compute and capacity of models has been increasing exponentially [1], with model training compute approaching 1000s of petaflop/s-days [31] and model capacity approaching trillions of parameters [86]. To train ever larger models, training clusters are scaling up to thousands of devices [76]. Efficiently parallelizing compute onto the devices in these large training clusters helps make training faster and resource efficient.

In this section, we give a short primer on the compute parallelization strategies used for ML training and the accompanying communication operations (comm-ops) that are issued. These comm-ops induce higher network traffic and we also give an overview of how the network infrastructure in training clusters is adapting to deal with it.

4.2.1 Parallelization Strategies and DLRM

We exemplify the different parallelization strategies via the Deep Learning Recommendation Model (DLRM). DLRM has been widely studied in industry and academia, and the largest DLRM used in production has trillions of parameters [42, 89, 86], making DLRM training especially challenging. DLRM uses a hybrid mix of parallelization strategies for different portions of the model. In fact, similar to DLRM, most state-of-the-art model families such as BERT [44], Megatron [96], GPT [31] are increasing in size and embracing hybrid parallel training.

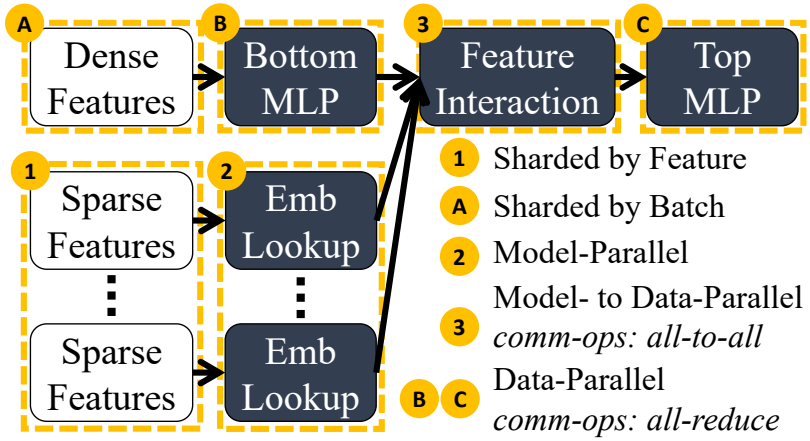


Fig. 4.1 DLRM Model

Figure 4.1 shows the DLRM model architecture. The training data comprises a mixture of dense continuous features and sparse categorical features (one-hot encoded or multi-hot encoded data), which are first mapped to a common embedding space using the bottom multi-layer perceptron (MLP) and the embedding table lookups respectively. The output embeddings go through a feature interaction phase and are then fed to the top MLP to get the recommender model output.

Data-Parallelism: Data-Parallelism is a common strategy for most Deep Neural Networks (DNNs) with parameters that can fit within a single device and with training dataset already sharded into batches across the devices. With data-parallelism, all the model parameters are replicated across all the training devices and each device has a worker process computing

parameter gradients in parallel. In the case of DLRM, the bottom MLP and top MLP use data-parallelism for training in production. These MLPs are compute intensive but not memory intensive and the MLP parameters fit within a single device memory. These MLPs are replicated across all the devices and the input data (continuous features) to these MLPs are already sharded into mini-batches.

Model-Parallelism: Data-Parallelism does not work for models with large capacity and with input datasets that cannot be trivially sharded into batches. With Model-Parallel training, the model is partitioned (and not replicated) across different devices. For DLRM, the embedding table lookup models convert the sparse categorical features stored in feature tables to an embedding. These table lookup models and the input tables are large and memory-intensive and do not fit a single device. As a result, these are parititioned across different devices during training resulting in model-parallel training.

Hybrid-Parallelism: As seen so far, different portions of DLRM training use different parallelization strategies. This is known as hybrid-parallelism. In the most general case, models can be replicated or partitioned in several different ways during training [71, 76, 88], resulting in hybrid-parallelism.

4.2.2 Communication Operations (Comm-Ops)

Different parallelism strategies induce different comm-ops.

All-Reduce: With data-parallel training, gradients computed at each worker process are exchanged over the network and aggregated to compute parameter updates. For example, during data-parallel training of layer-by-layer models, each layer triggers such a pattern of communication during backward propagation. This pattern of communication is known as the all-reduce collective.

All-To-All: After the embedding lookups in DLRM (② in Figure 4.1), each device has a

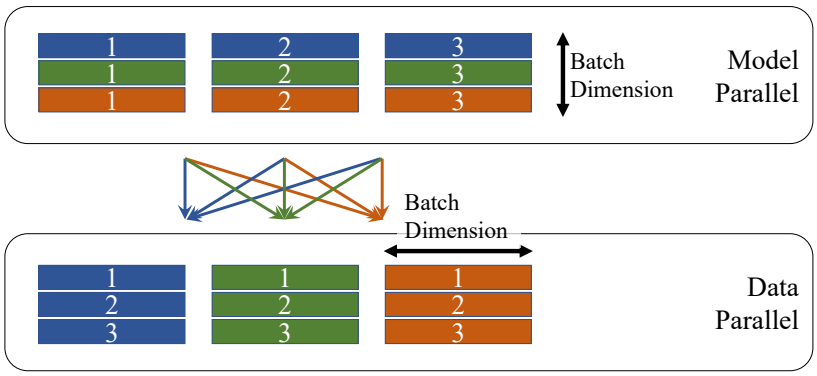


Fig. 4.2 Illustration of all-to-all collective in DLRM

vector for the table lookup models resident on those devices for all the samples in the batch, which needs to be reorganized and sharded along the batch dimension. This induces an all-to-all pattern of collective communication, as shown in Figure 4.2.

Collectives from the MPI standard [84]: In the general case, hybrid-parallel training results in several types of comm-ops, ranging from all-reduce, all-to-all to collective-permute, all-gather for certain models [76] to any collective defined in the MPI standard [84].

4.2.3 Evolving Network Infrastructure

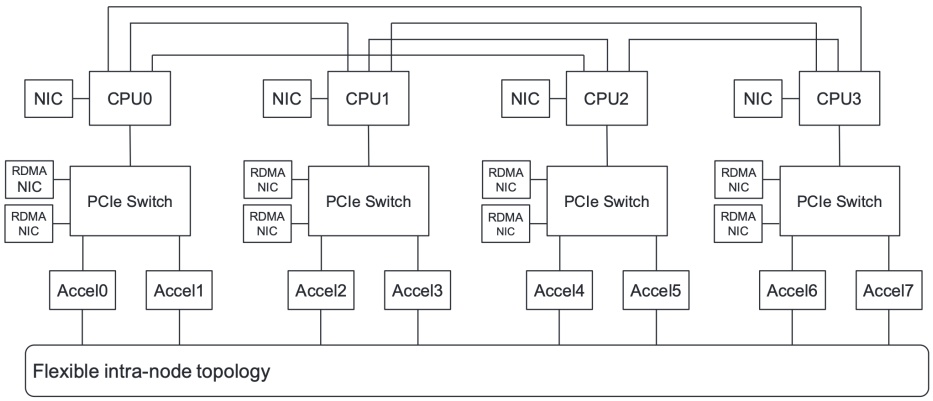
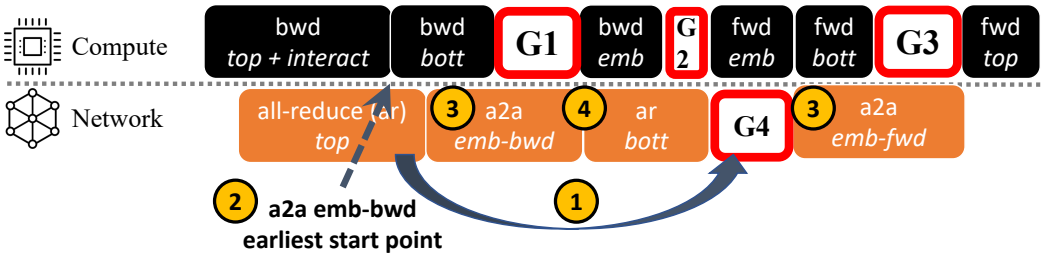


Fig. 4.3 State-of-the-art system architecture of training cluster [86]

The aforementioned comm-ops push increasing amounts of network traffic and the network infrastructure is adapting with fatter topologies and faster interconnects to ensure the needed throughput and latency. We exemplify this via the networking infrastructure in a state-of-the-art training cluster according to [86] as shown in Figure 4.3. Each node has 4-socket CPUs and 8 GPUs, with 4 frontend NICs connected to the host CPUs and a dedicated RDMA over Converged Ethernet (RoCE) backend NIC for each of the GPUs connected via PCIe switches. The backend NICs from across nodes can be connected with a dedicated backend network to form a training cluster. The RoCE NICs allow for more efficient RDMA transfers. The extensible design of this node allows to scale-out the backend network to interconnect thousands of nodes, forming a data-center scale AI training cluster. This training cluster has heterogeneous mix of networking interconnects and protocols to drive network traffic with varying throughput and latency guarantees. This heterogeneity leads to the presence of multiple communication channels between any two endpoints. At an intra-node level, pair of GPUs can communicate via shared memory, NVLink complex, PCIe complex or the external network. At an inter-node level, any two GPUs can communicate via GPUDirect RDMA [90] or TCP/IP over Ethernet.



Compute Stream Gaps = G1+G2+G3 = 13.6% + 5.4% + 15.8% = 34.8%
Communication Stream Gap = G4 = 6.3% of iteration time

Fig. 4.4 Gaps in DLRM Trace

4.3 Motivation

As exemplified by DLRM, emerging ML training deployments are trending towards larger models and synchronous hybrid-parallel training that issue various types of comm-ops i.e., collectives over the network.

Communication Bottleneck: Despite the networking infrastructure upgrades, the execution of these comm-ops are a source of excessive delays in compute. Figure 4.4 illustrates the execution of Compute Unified Device Architecture (CUDA) stream kernels on a randomly chosen GPU during a single iteration of production scale DLRM training [1]. The DLRM training creates a compute and a communication stream for serialized execution of tensor operator kernels and comm-op kernels, respectively. As emphasized in the figure, we note that there are several gaps during execution. A gap on a stream occurs when the stream is waiting for the result of kernel execution on the other stream. The compute stream gaps are wider (34.8%) and cumulatively larger than those in the communication stream (6.3%). This means that communication is a training bottleneck as it blocks compute for a third of the iteration. We now show that there are several opportunities to optimize comm-ops.

Better Scheduling Opportunity: Reordering of comm-ops can lead to greater overlap of compute and communication and reduce gaps. We highlight this in Figure 4.4 – ①: the top MLP all-reduce comm-op can be split judiciously and partially executed later to occupy the gap G4; ②: as a result the all-to-all backward comm-op can be pulled up to begin as soon as possible to reduce the gap G1.

Better Execution Planning Opportunity: Existing comm-ops do not efficiently utilize multiple communication channels available in heterogeneous network interconnects. We highlight this in Fig.4.4 – ③: both the all-to-all's can be broken up into smaller fragments of communication work and executed one fragment at a time to reduce incast and improve throughput to reduce gaps G1 & G3; ④: all-to-all and all-reduce can be executed in parallel

[1] We show accurate percentages and hide low-level details.

over communication channels with non-overlapping interconnects to start all-reduce sooner and drive higher network throughput to reduce gap G2.

Several works look to reduce communication overheads via smarter scheduling and efficient execution planning. However, existing works are optimal for specific training scenarios (PS architecture [59], layer-by-layer models [94], all-reduce collectives [72, 113] as detailed in §4.7); and the scheduling and execution planning techniques proposed therein are ad-hoc as they have restrictive assumptions and making it unclear as to how to compose and apply these different techniques towards hybrid-parallel training of DLRM-like models.

There is room for improvement in existing scheduling and execution planning techniques. However, we argue that, a more fundamental shortcoming of these works is that they do not explore *joint optimization* (§4.3.1) mainly because *existing interfaces in the communication stack used for ML training are not naturally amenable* (§4.3.2). We also note that *a collective is often too coarse-grained* to schedule communication work; breaking it up improves communication optimization flexibility (§4.3.3). We start by first giving an overview of the existing communication stack used for ML training.

4.3.1 Disjoint Scheduling, Execution Planning

4.3.1.1 Communication Stack Overview

Figure 4.5 shows the two sets of layers in the communication stack used for ML training: the application (app) layer and the communication (comm) layer. The app layer and the comm layer is representative of the set of concerns managed by ML training frameworks (such as TensorFlow, PyTorch) and the communication libraries (such as MPI, NCCL, UCC), respectively. Figure 4.5 shows the four steps leading to execution of a comm-op over the network –

① **Model Definition:** The user defines a model by composing various tensor operations. The example shows a model declaration with three operators and its tensor operator graph.

② **Parallelization Strategy:** The parallelize module (e.g., `nn.DistributedDataParallel` in PyTorch) takes the model and the set of devices and converts the computation graph to a training DAG. Before training begins, the parallelize module statically decides: the parallelization strategy i.e., how to split the dataset, the model parameters across tasks during forward and backward propagation; and the placement of these tasks i.e., the devices these tasks execute on. This training DAG is repeatedly executed every training iteration. The tasks are compute-ops or comm-ops and edges capture dependencies. The above example shows the training DAG for a single iteration and the ops in the training DAG are managed

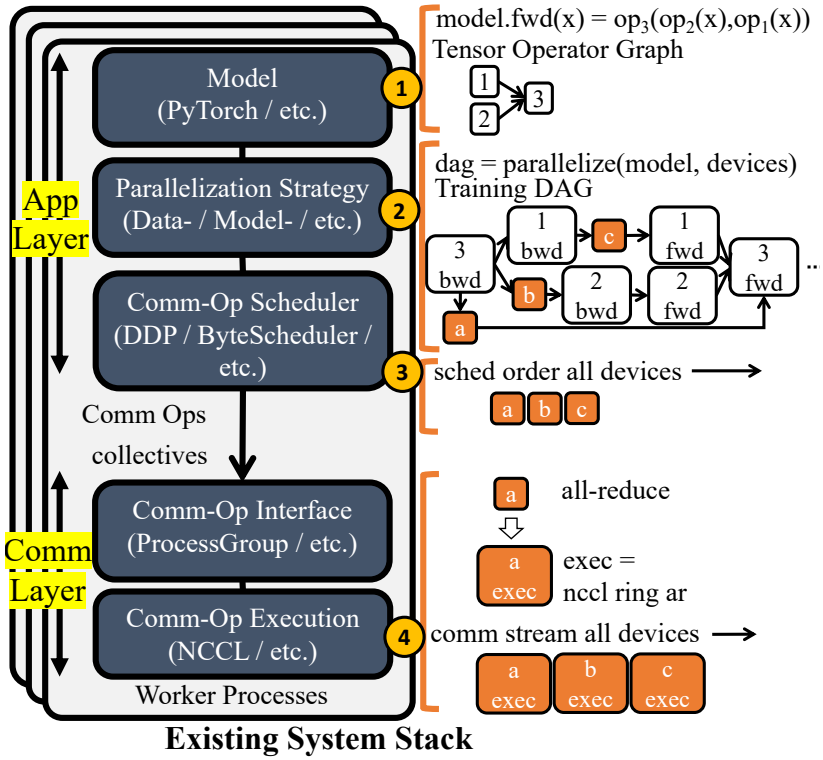


Fig. 4.5 ML Training Communications Stack

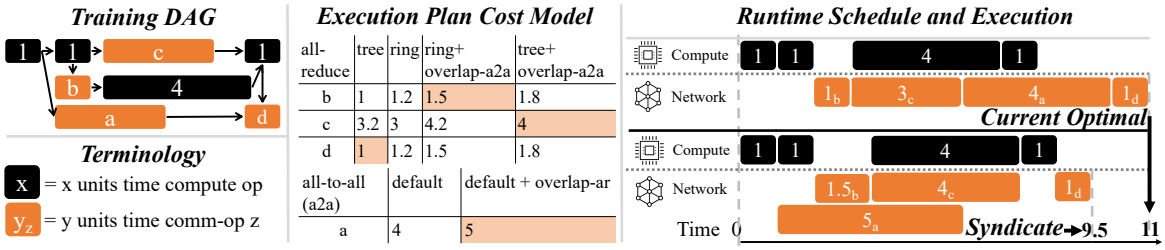


Fig. 4.6 Motivating Example

by spawning several worker processes on all the devices by using a parallel programming library such as MPI.

In this work, we do not optimize the parallelization or placement strategy in the training DAG as several works have explored that aspect [71, 76]. We exclusively focus on rethinking scheduling and execution planning.

③ **Comm-Op Scheduling:** The communication scheduler takes the training DAG as input and decides the optimal ordering for the comm-ops. The scheduler chooses an order that maximizes overlap of compute and communication. The scheduling procedure is deterministic and executes on all the worker processes so that the comm-ops are issued every training iteration in *the same order on all the devices*. The example shows the default PyTorch order i.e., FIFO. The app layer at all the devices submits these comm-ops in this same order to

the comm layer for execution. This ensure sthat the order is deterministic and co-ordinated across all devices so that all the worker processes execute comm-ops in the same order.

④ **Comm-Op Execution Planning:** The comm layer at all the devices receive the comm-op from the app layer via an interface with a well-defined API (e.g., `ProcessGroup` in PyTorch). The execution planner on receiving each comm-op, assigns it an execution plan and queues it in the same order on the devices' i.e., GPU's communication stream for serialized execution. The example shows an all-reduce collective which binds to NCCL's ring all-reduce implementation during execution. Each collective typically has several options for its execution plan (e.g., ring or tree for all-reduce collective) and execution planners, such as NCCL, have network topology aware cost models that estimate the execution time for different options. Current execution planners are greedy and select the execution plan option with the least execution time. This greedy procedure runs on all the worker processes and since it relies on static inputs such as topology, it is deterministic and this ensures that collectives issued on all the worker processes bind to the same execution plan.

Scheduling and Execution Planning Today: As we just described, scheduling is an app-layer concern governed by schedulers in ML training frameworks as PyTorch, while execution planning is a comm-layer concern governed by execution planners in communication libraries as NCCL. As these concerns are not jointly optimized and are taken independently, several inefficiencies arise, which we highlight next through an example.

4.3.1.2 Example to highlight suboptimality

Figure 4.6 illustrates the lost opportunities due to a lack of joint optimization of scheduling and execution planning. The network topology is similar to that illustrated in Figure 4.3. The training DAG in this example has four collectives: `a` is an all-to-all collective and `b`, `c`, `d` are all-reduce collectives. There are several execution plan options for each collective. There is a cost associated with each option which measures the execution time over the network. At the granularity of a single comm-op, lower cost is preferred and means faster execution. For the case of the all-reduce collective, the execution plan options are tree all-reduce or the ring all-reduce, both using NVLink and GPUDirect RDMA. For the case of the all-to-all collective, there are two execution plan options: pairwise exchange between all processes either using NVLink and GPUDirect RDMA or using PCIe complex and TCP/IP over ethernet. With the latter option for all-to-all, all-to-all and all-reduce can be overlapped. We compare the iteration time of the current solution against SYNDICATE.

Current Solution: The execution planner greedily selects the fastest option for each collective. The scheduler first estimates the comm-op execution time by modeling greedy behavior of the execution planner or by benchmarking the model for a few iterations. The scheduler

then uses DAG scheduling algorithms to find the optimal comm-op scheduling order. This results in a training iteration time of 11 units.

SYNDICATE Solution: SYNDICATE realizes that by jointly making changes to the scheduling order and execution plan choices there is opportunity to overlap all-to-all with all-reduce and speed-up communication by utilizing network heterogeneity. The current solution's scheduling order executes all-to-all last and does not allow overlap. SYNDICATE's scheduling order executes all-to-all collective at the very beginning and allows overlap. The execution plan choices made by SYNDICATE are shaded in the execution plan cost model in Figure 4.6. SYNDICATE's execution planner is not greedy and chooses a slower execution plan for all-to-all so as to allow for parallel execution of all-to-all and all-reduce over non-overlapping interconnects in the network. Overall, this results in a training iteration time of 9.5 units and is faster than the current solution.

Joint optimization is beneficial but current interfaces are not amenable as we discuss next.

4.3.2 Interface constraints Joint Optimization

The training DAG scheduler in the ML processing frameworks is unaware of optionality (e.g., an all-reduce can be executed by as a ring all-reduce or a tree all-reduce) present lower down the stack during execution planning. A trivial extension of the existing interface is to expose the training DAG to the comm layer and push the scheduling concern down the stack to co-locate it with the execution planner. Exposing the training DAG down the stack is necessary to ensure that any reordering of comm-ops down the stack does not lead to *dependency violations*: a child comm-op cannot be ordered before a parent comm-op as otherwise it can lead to a *deadlock*. This enables joint optimization without dependency violations. However, the joint optimization problem is NP-hard and the joint optimization procedure requires a time-intensive, randomized algorithm (§4.4.3). As a result, this procedure can delay comm-op execution due to its time-intensive nature. To make matters worse, this randomized procedure, may lead to divergent scheduling orders across different processes. This can lead to *out-of-sync* issues, wherein if the collectives are not submitted in the same order across two different processes then it results in a *deadlock* where each process waits for the other process to issue the same collective as itself. As a result, the existing interfaces are unable to trivially accommodate joint optimization of these concerns.

4.3.3 Issues with Coarse-Grained Scheduling

Scheduling today happens at the granularity of user-submitted comm-ops i.e., collectives. Communication libraries such as NCCL, submit each comm-op as a kernel on the GPU

communication stream and a kernel cannot be context-switched during execution. This means that once a comm-op is scheduled for execution it cannot be stopped mid-execution. This leads to limited scheduling flexibility in space and time.

Each comm-op has two attributes: a payload and a pattern of network transfers. If the payload is very large then each network transfer in the comm-op, once scheduled for execution, occupies the network links for a long time. If the pattern of network transfers is large (e.g., a clique of network transfers between all pairs of workers in the case of an all-to-all collective) then the comm-op, once scheduled for execution, gang schedules transfers on a large fraction of network interconnects. Comm-ops, if scheduled as-is, thus have large communication work orders and limit the ability to context switch communication work in time and efficiently pack communication work in space i.e., over all the heterogeneous interconnects available in the network.

4.4 SYNDICATE Design

SYNDICATE changes the interfaces in the communication stack to enable joint optimization of scheduling and execution planning. It builds on the *motif* abstraction to enable deconstructing comm-ops into smaller work units along a few dimensions and allow finer-grained scheduling. In this section, we start with an overview of the new interfaces and the new modules in SYNDICATE's communication stack and how it enables joint optimization (§4.4.1). We then explain the motif abstraction (§4.4.2), the joint optimizer design (§4.4.3), and enforcement of the joint optimizer's decisions (§4.4.4).

4.4.1 Overview

Figure 4.7 shows SYNDICATE's communication stack. Notably, we propose two new entities: a central optimizer and an enforcer. SYNDICATE co-locates scheduling and execution planning concerns in the centralized joint optimizer. The central optimizer generates an optimizer plan. This plan contains instructions on how to execute as well as how to schedule the comm-ops during training and is conveyed to the enforcer on each worker process. In this regards, *the central optimizer is the the control plane while the enforcer is the data plane*. We propose interfaces (Ⓐ, Ⓑ, Ⓒ) between the central optimizer and the communication stack. These interfaces are out-of-band and asynchronous, meaning that the data plane does not, in any circumstances, block execution of a comm-op waiting for control plane instructions.

We now go over the workflow in SYNDICATE. We divide it into control plane workflow and the data plane workflow.

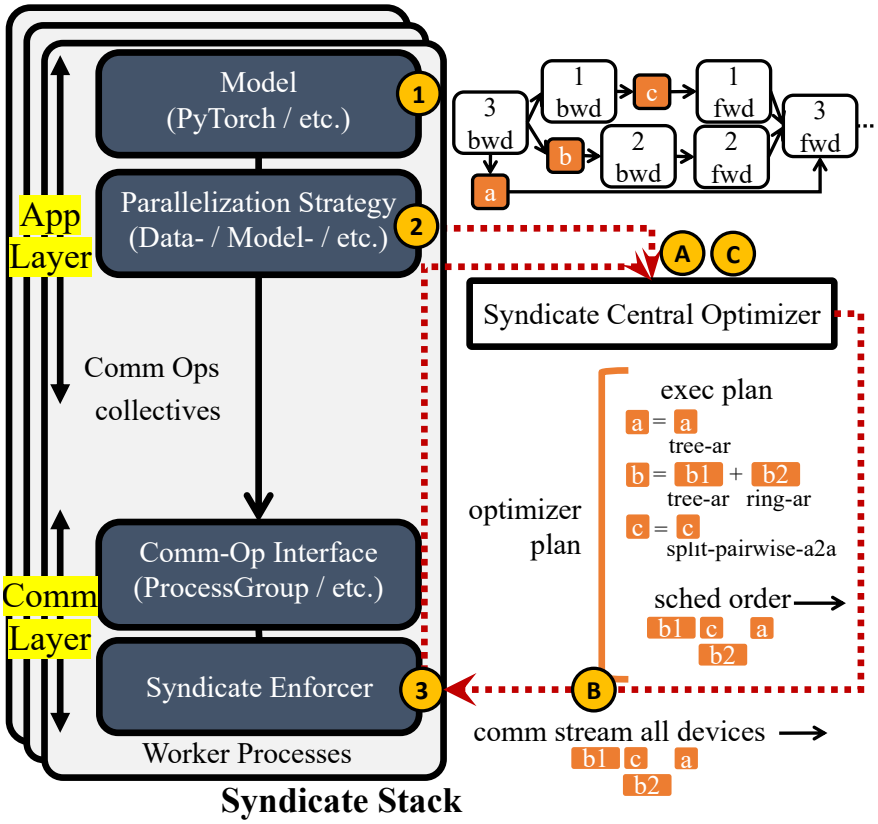


Fig. 4.7 Overview of SYNDICATE's ML training Communication Stack

Stepping through the control plane workflow –

A Joint Optimization: The central optimizer pulls the training DAG from the app layer and the network topology from the comm layer. The optimizer uses these inputs to construct the execution plan cost model and does joint optimization to yield the optimizer plan (§4.4.3).

B Optimizer Plan Distribution: The joint optimizer plan is sent to the enforcer on all the worker processes (§4.4.3).

C Feedback: The central optimizer pulls comm-op performance statistics from the enforcer to help refine the execution plan cost model and potentially redo joint optimization (§4.4.4).

Stepping throught the data plane workflow –

1 Model Definition: The user defines a model by composing tensor operators. This yields a computation graph (§4.3.1).

2 Parallelization Strategy: The computation graph is converted to a training DAG (§4.3.1). The comm-ops from the training DAG are submitted every training iteration as-is to the comm layer in the default FIFO order without applying any scheduling optimizations.

3 Optimizer Plan Enforcer: The comm-ops are executed as instructed by the central optimizer (§4.4.3).

4.4.2 Motif Abstraction

A motif is a logical grouping of several point-to-point transfers over the network. The enforcer schedules and executes communication work at the granularity of motifs. A motif once issued to the device e.g., as a kernel on GPU communication stream is non-preemptible and occupies network resources until the communication work in the motif is completely executed.

Conversion of comm-op to motifs: Each comm-op i.e., a collective has two attributes: a payload (typically tensors) and a pattern of network transfers. We propose two transformation operators to slice a comm-op either along the payload dimension or the pattern dimension into one or more motifs. As compared to the original comm-op, each motif represents a smaller unit of communication work (with reduced payload size and/or smaller pattern). Since SYNDICATE does scheduling at the granularity of motifs, this enables finer-grained scheduling with increased opportunities for making more frequent scheduling decisions in time to enable better overlap of compute/communication and packing smaller units of communication work more efficiently over the network resources.

4.4.2.1 Motif Transformation Operators

Segmentation and Splining: SYNDICATE proposes two transformation operators: segmentation and splining. Segmentation splits the payload into smaller payload segments. Splining splits the pattern of network transfers into smaller patterns. Figure 4.8 and Figure 4.9 illustrates these operators.

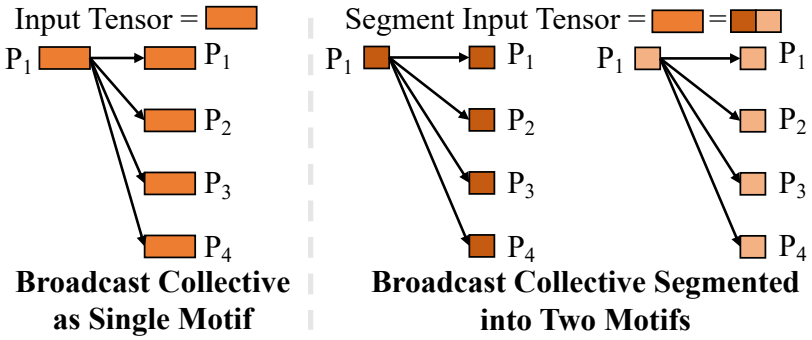


Fig. 4.8 Example showing segmentation of broadcast collective. Left half shows the broadcast collective as a single motif. Right half shows broadcast collective segmented into two motifs, where each motif broadcasts one half of the bytes from the original tensor.

Transformation Operator Algebra: We now formalize the algebra for the motif transformation operators. The goal of this algebra is to state concrete rules for transforming comm-ops

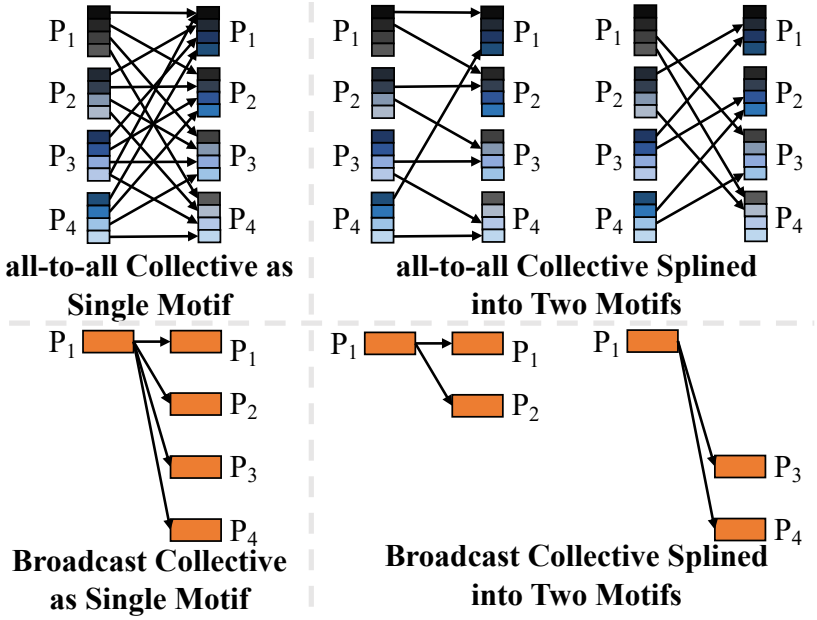


Fig. 4.9 The left half shows all-to-all and broadcast collective as a single motif that bundles the transfers from all the source processes to all the destination processes. The right half shows both the collectives splined into two motifs, where each motif transfers the same payload from the source process to one half of the destination processes.

into motifs. This formalization succinctly encodes: (1) correct and admissible motif transformations, (2) correct and admissible transformation combinations, and (3) a structured space for all possible operator compositions. We denote the segmentation operator by $\stackrel{s}{=}$ and the splining operator by $\stackrel{p}{=}$. These rules are by no means exhaustive and are extensible. We first present the symbols used in the algebra.

$\|$: Parallel Execution Permitted

$\stackrel{s}{=}$: Segmentation Transformation

$\stackrel{p}{=}$: Splining Transformation

`N` : Total number of Processes

`PG[IDs]` : Process `IDs` involved in a Collective

T_i`[0:N,0:D]` : `N` Tensors of size `D` on Process P_i with
T_i`[j,0:D]` destined for Process P_j

M_{AA}`(`T_i`[0:N,0:D], PG[0:N])` : collective with all-to-all pattern of transfers
with tensor T_i as payload
executing on each Process P_i for all i in `0:N`

`M(`T_i`[`a_i`:`b_i`,`x_i`:`y_i`], PG[IDs])` : Motif `M`
executing on each Process P_i for all i in `IDs`
with payload = tensor T_i`[j,` x_i`:`y_i`]` from Process P_i
destined to Process P_j for all j in a_i`:`b_i

Next, we present the algebraic rules that we use in the context of DLRM to transform all-to-all into motifs. The algebra for other comm-ops is in the Appendix §C.0.1.
Segmented All-To-All:

$$\texttt{M}_{AA}\texttt{(T}_i\texttt{[0:N,0:D], PG[0:N])} \overset{\text{S}}{=} \|_{s=0}^{\frac{D}{d}-1} \texttt{M(T}_i\texttt{[0:N, s*d:(s+1)*d], PG[0:N])}$$

With segmentation, the payload to be sent from a source process to all the destination processes is split into segments of size `d` (= $\texttt{T}_i$`[0:N, s*d:(s+1)*d]`). Segmentation of all-to-all in this way, yields $\frac{D}{d}$ motifs where each motif sends a payload of size `d` from a source process keeping the set of destination processes the same. `d` is a parameter and controls the number of motifs associated with the input all-to-all.
Splined All-To-All:

$$\texttt{M}_{AA}\texttt{(T}_i\texttt{[0:N,0:D], PG[0:N])} \overset{\text{p}}{=} \|_{c=0}^{\frac{N}{n}-1} \texttt{M(T}_i\texttt{[(i+c*n)\%N:(i+(c+1)*n)\%N, 0:D], PG[0:N])}$$

With splining, the pattern of network transfers in the all-to-all with each source process sending the payload to all the `N` destination processes is broken down into smaller patterns, where each source process $\texttt{P}_i$ sends the same payload as before to `n` destination processes (= `(i+c*n)%N:(i+(c+1)*n)%N`). Splining of all-to-all in this way, yields $\frac{N}{n}$ motifs. Here, `n` parameterizes the all-to-all splining operator with larger `n` breaking the all-to-all into fewer motifs with larger sub-patterns.
Composition of Operators: Note that the all-to-all collective, $\texttt{M}_{AA}\texttt{(:)}$, is in fact a special case single motif (with `d` = `D` and `n` = `N`). These operators can be composed and recursively break a motif into several more finer-grained motifs. While fine-grained motifs are beneficial for scheduling flexibility, there is a fixed overhead associated with dispatching a motif as a kernel on GPU communication stream and launching it during execution and too fine-grained motifs are not desirable as these overheads can slow-down communication.
Physical Plan for Motif: Each motif bundles together several network transfers. Physical plan determines the physical interconnects that each network transfer is assigned to. Figure 4.10 shows an example of a physical plan for the broadcast collective. The broadcast collective is first broken into three motifs. The physical plan maps the motif to point-to-point network transfers over various interconnects available in the network. The figure also shows a toy network topology where the GPU for process $P1$ connects to all other GPUs via both PCIe and NVLink interconnects. The three motifs can be multiplexed over different interconnects. The physical plan for the first motif does a `memcpy` on process $P1$. The physical plan for the remaining two motifs use PCIe and NVLink in a mutually exclusive manner. This allows the

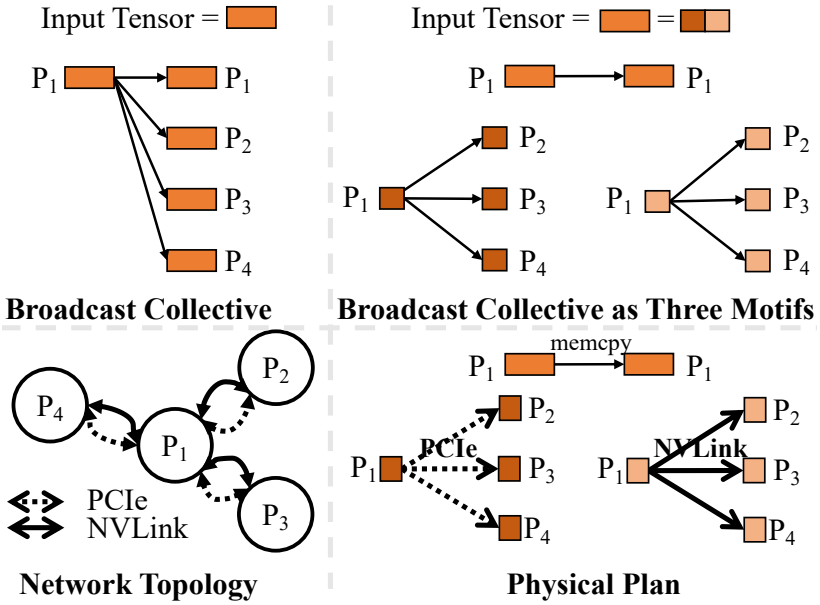


Fig. 4.10 Physical Plan for Broadcast Collective

point-to-point transfers in the three motifs to execute in parallel, maximizing utilization of multipath opportunities available in the network.

4.4.3 Central Optimizer

The central joint optimizer is responsible for minimizing training iteration time by minimizing communication overheads. The optimizer determines the optimizer plan by systematically navigating the vast space of schedules for different execution planning options.

The optimizer plan has two pieces: the execution plan and the scheduling order, containing instructions regarding how to execute and how to schedule comm-ops respectively. The execution plan transforms each comm-op in the training DAG into one or more motifs. The scheduling order decides the order of execution of motifs.

Exponential Search Space: There is a lot of optionality in the execution plans for each comm-op. As we saw before, the transformation operators can be composed to break a comm-op into motifs in several different ways. Let us say that there are atleast O execution plan options for each comm-op and there are C comm-ops in the training DAG, then this results in O^C unique execution plan options for all the comm-ops in the training DAG.

Cost of each Execution Plan: For a particular execution plan, there is an optimal scheduling order for the motifs that maximizes overlap of compute/communication and minimizes training iteration time. This training iteration time with the optimal scheduling order is the cost of the execution plan. Note that finding the optimal scheduling order can be reduced to a general DAG-scheduling problem and has no known polynomial time algorithm.

Pseudocode 4 Probabilistic Search

```
1: Training DAG with Greedy Execution Plan G_*

2: procedure MCMCSEARCH
3:     C_*, sched_order_* = optSched(G_*)
4:     while true do
5:         G_temp = transform(G_*)                          ▷ change execution plan for a comm-op at random
6:         C_temp, sched_order_temp = optSched(G_temp)
7:         α(C_temp | C_*) = min(1, exp(β * (C_* - C_temp)))
8:         G_*, C_*, sched_order_* = G_temp, C_temp, sched_order_temp with α prob.
9:     return G_*, sched_order_*

10: procedure OPTSCHED
11:     comm_q                                               ▷ queue of ready communication motifs
12:     compute_q                                            ▷ queue of ready compute tasks
13:     comp_time = 0
14:     comm_time = 0
15:     sched_order
16:     while comp_time ≤ comm_time and comp_q != φ do
17:         comp_task = fifoDequeue(comp_q)
18:         comp_time += comp_task.time()
19:         sched_order.schedule(comp_task)                  ▷ enqueue ready motifs, compute
20:     while comm_time ≤ comp_time and comm_q != φ do
21:         comm_motif, startTime = criticalPathDequeue(comm_q)
22:         comm_time = max(comm_time, startTime+comm_motif.time())
23:         sched_order.schedule(comm_motif)                 ▷ queue ready motifs, compute
24:     return max(comm_time, comp_time), sched_order
```

Problem Statement: The centralized joint optimizer takes a training DAG G and the network topology as inputs. We take a training DAG that unrolls compute-ops and comm-ops across two training iterations to enable cross-iteration optimizations. The aim of the joint optimizer is to take these inputs and find the execution plan with minimal cost. The joint optimizer outputs the optimizer plan, which has the execution plan and the scheduling order that minimizes overall cost.

4.4.3.1 Joint Optimization Procedure

The key idea in SYNDICATE is to do *probablistic search* over the exponentially large search space. We use MCMC search as outlined in Pseudocode 4.

MCMC Search: The joint optimizer starts with the default execution plan for the training DAG (denoted by G_*), wherein all the comm-ops are greedily assigned the execution plan choice with the minimum possible execution time. Thereafter, a comm-op is chosen at random and it is assigned a random execution plan option. This changes the motifs associated with this particular comm-op, keeping all the other motifs constant and yields a temporary execution plan for the training DAG (denoted by G_{temp}). The cost i.e., the execution time of this training DAG is calculated using the optSched (line 11 in Pseudocode 4) procedure which is a greedy scheduling heuristic to always dequeue motifs on the critical path in the DAG to maximize overlap of communication motifs with compute tasks or other communication motifs (in case the two communication motifs have non-overlapping physical plans). This update to

Pseudocode 5 Distributed Optimizer Plan Enforcer

```
1: Exec Plan 𝔼_colls = {..., coll_i^in : {motif_{i,j}^out}, ...}        ▷ optimal execution plan
2: Scheduling Order 𝕊 = {..., motif_{i,j}^out : seq_{i,j}^num, ...}     ▷ optimizer scheduling order
3: Progress Queue pq                                                    ▷ thread-safe priority queue containing ready motifs

4: procedure ENFORCEEXECPLAN(coll^in)                                   ▷ app submits comm-op to comm layer
5:     {motif^out} = 𝔼_colls[coll^in]                                   ▷ comm-op is deconstructed into motifs
6:     for all motif^out ∈ {motif^out} do
7:         seq^num = 𝕊[motif^out]
8:         pq.INSERT(priority=seq^num, motif^out)

9: procedure ENFORCEORDER                                               ▷ runs in a separate thread and enforces order
10:    nextMotifSeqNum = 0
11:    while true do
12:        while pq.TOP().priority != nextMotifSeqNum do
13:                                                                     ▷ busy loop until next in order motif is ready
14:        nextMotifSeqNum += 1
15:        {motif} = pq.POP()
16:        {motif_tensors} = {motif}.EXECUTE()
17:        REPACK({motif_tensors})
```

the execution plan is probablistically sampled using the Metropolis-Hastings algorithm [60] and retained in $\mathtt{G}_*$ (line 8 in Pseudocode 4). This tends to behave as a greedy search over the search space with an ability to escape local minimas [60, 71].

Search Termination: MCMC search is terminated if the search procedure exceeds the time budget assigned for search or if the search procedure does not find a better joint optimizer plan for more than half of the total elapsed search time.

4.4.4 Enforcer

The central optimizer commits the same joint optimizer plan, comprising of the execution plan and the scheduling order to the enforcer on each worker process. The enforcer is responsible for *tightly co-ordinating this plan across all the worker processes during training* so as to avoid deadlocks and out-of-sync issues (§4.3.2). The application thread spawned by the ML processing framework at each worker process submits comm-ops to the comm layer every training iteration. With SYNDICATE, these comm-ops are submitted one-at-a-time in FIFO order. These comm-ops are intercepted by the enforcer. The enforcer is responsible for execution of these comm-ops and preparing the result of these comm-ops (tensors) to unblock the next application thread operation (compute-op or comm-op typically waiting on a CUDA stream) that is waiting on these tensors.

The enforcer takes the responsibility of executing these comm-ops as per the instructions of the optimizer plan and preparing the output tensors once ready. It does so in three steps. First, on intercepting a comm-op, it enforces the execution plan by breaking it into motifs. Second, it enforces the desired scheduling order of execution of motifs. Third, as and when motifs complete, it checks for completion of comm-ops and packages the output of individual

motifs into the comm-ops output tensors. Pseudocode 5 shows the procedure to enforce the execution plan and the scheduling order contained in the joint optimizer plan. The repacking of tensors to comm-op output tensors happens after successful execution of each motif (line 20 in Pseudocode 5).

Enforcing Execution Plan: The enforcer is layered as a shim on top of existing comm-op execution layer i.e., different communication libraries such as NCCL, MPI, UCX. The app layer submits comm-ops to the comm layer using the interface between them and is immediately trapped by the `enforceExecPlan` procedure (line 4 in Pseudocode 5). This procedure deconstructs the comm-op to one or more motifs as dictated by the execution plan in the joint optimizer plan. Each motif is also assigned a sequence number. The sequence number captures the priority of this motif in the current training iteration. This sequence number is contained in the scheduling order of the joint optimizer plan. These motifs are enqueued into a priority queue using the sequence number as the priority.

Enforcing Scheduling Order: The `enforceOrder` procedure (line 11 in Pseudocode 5) enforces the scheduling order and runs in a thread separate from the `enforceExecPlan` procedure. This procedure maintains a priority counter that is incremented sequentially and is reset at the end of each training iteration. This counter maintains the priority of the next expected motif(s). In case of overlapping motifs, two or more motifs can be assigned the same priority. The `enforceOrder` procedure busy loops until the priority of the motif at the top of the priority queue matches the value in the priority counter. It busy loops until the `enforceExecPlan` enqueues the next expected motif. This ensures that the enforcer on all the worker processes executes motifs in the same order.

Replanning: We measure the wait time in the busy loop and send it as feedback to the central optimizer. If wait times in every iteration consistently add up to more than a threshold (= 5% of iteration time), then we redo joint optimization at the optimizer to explore a different optimizer plan.

In case of failure of the central optimizer or delays in receiving instructions from the central optimizer, the enforcer gracefully switches to the default fallback i.e., executing comm-ops as-is, one-at-a-time and in-order (similar to today).

4.5 Implementation

We implement the central optimizer as a separate module in python. The central optimizer simulates the execution of different execution plan choices as part of the MCMC search procedure until the search procedure terminates and yields a joint optimizer plan. The central optimizer runs on one of the machines in the training cluster and interacts with the various

Compute (TFLOPS)	120 (FP32)/ 1000 (FP16)
HBM	256 GB, 7.2 TB/s
DDR	1.5 TB, 200 GB/s
Scale-up bandwidth	1.2 TB/s (uni-directional)
Scale-out bandwidth	8 x 100 Gbps (uni-directional)
Host NW	2 × 100 Gbps

Table 4.1 Configuration of each node in our cluster

enforcers on the worker processes via RPCs. We build a two-phase commit (2PC) protocol using RPCs so that the same joint optimizer plan is safely committed by the central optimizer to all the enforcers. After the 2PC protocol is complete, the enforcers switch to the new joint optimizer plan from the subsequent training iteration. This ensures that out-of-sync issues are avoided whenever transitioning to a new joint optimizer plan.

We implement the enforcer in the Unified Collective Communication (UCC) library interface [20]. We implement the enforcer routines: `enforceExecPlan` routine in the `main thread` and the `enforceOrder` routine in the `progressLoop thread` in the `torch-ucc` interface [16].

4.6 Evaluation

4.6.1 Testbed

We experimented with our prototype on a production-scale cluster using off-the-shelf NVIDIA HGX-2 based systems. Specifically, each node hosts dual-socket CPUs, 8 V100 GPUs that are fully-connected using NvSwitch, 2 front-end host NICs, and 8 back-end RoCE NICs to allow direct RDMA between GPUs on separate nodes. Table 4.1 shows the node configuration; we have 16 such nodes in our testbed.

The testbed is running CentOS-8 and CUDA 11.4 with NVIDIA driver 470.57.02. For distributed training of DLRM models, we used PyTorch 1.10 (nightly) with extension of Process Group UCC [16] and latest UCC library [20], which can take advantage of various transports such as NCCL 2.10.3 [12] and UCX-based [101] for dynamically selecting optimal execution planing of collective operations as SYNDICATE desired.

4.6.2 Workloads

We report results for performance of three DLRM models. Table 4.2 shows the details. We have progressively wider and higher number of embedding tables from model A1 to A3. These embedding tables are not compute-intensive and do not contribute to increasing the

Model	A1	A2	A3
Num parameters	95B	793B	845B
MFLOPS per sample	89	638	784
Num of emb tables	$\sim$ 100s	$\sim$ 1000s	$\sim$ 1000s
Emb table dim	[4, 192]	[4, 384]	[4, 960]
(range [min, max], avg)	avg: 68	avg: 93	avg: 231
Avg pooling size	27	15	17
Num MLP layers	26	20	26
Avg MLP size	914	3375	3210

Table 4.2 Models in our workload

MFLOPs per sample but have a high memory footprint (and contribute to higher number of parameters) and induce progressively more communication bandwidth-hungry all-to-all's to transfer a large number of embeddings. We also have progressively wider MLPs from model A1 to A3, which are compute intensive and increase the model compute (MFLOPS per sample).

4.6.3 Metrics

We measure the following metrics (1) **Training Throughput:** We measure the training throughput in terms of recommendation queries per second (QPS). Higher QPS is desirable. (2) **Compute Idling:** We measure the idle gaps in the GPU's compute stream and report it as a percentage of the total iteration time. Lower compute idling is desirable. (3) **Normalized Metric:** We normalize the metric (such as QPS) against baseline using the formula – $\frac{\text{Metric with SYNDICATE}}{\text{Metric with Baseline}}$

4.6.4 Baselines

We compare SYNDICATE against the following baselines.

- Last-In-First-Out Scheduler (LIFO): A recent scheduler, ByteScheduler [94], proposes LIFO scheduling policy for maximizing overlap of compute and communication. ByteScheduler implementation only supports all-reduce in layer-by-layer models and does not have support for all-to-all collectives. We emulate ByteScheduler via our own implementation of the LIFO scheduler that is co-located with PyTorch framework. To emulate the bayesian optimizer used in ByteScheduler to optimally segment individual send operations, we aid our LIFO scheduler with a segment oracle that optimally segments all collectives.

- Hand Optimized Model (HO): Existing execution planners do not optimize all-to-all collectives and existing schedulers do not have support for DLRM-like models. We hand optimize the scheduling policy in PyTorch (and also provide it the benefit of the segment oracle) and hand optimize the choice of execution plan for each collective (including all-to-all) in the UCC communication library. In this regards, HO models the best possible solution with today's placement of scheduling and execution planning concerns in exiting stacks.
- SYNDICATE-Exec (S-Exec): We disable scheduling optimizations in SYNDICATE. We do so by using PyTorch's default scheduler that does FIFO scheduling to estimate the cost of each execution plan during joint optimization.
- SYNDICATE-Sched (S-Sched): We disable execution planning optimizations in SYNDICATE. We do so by disabling the MCMC search procedure and find the optimal scheduling order using SYNDICATE's scheduling heuristic (and also provide it the benefit of the segment oracle to optimally segment collectives).

4.6.5 DLRM Evaluation on Testbed

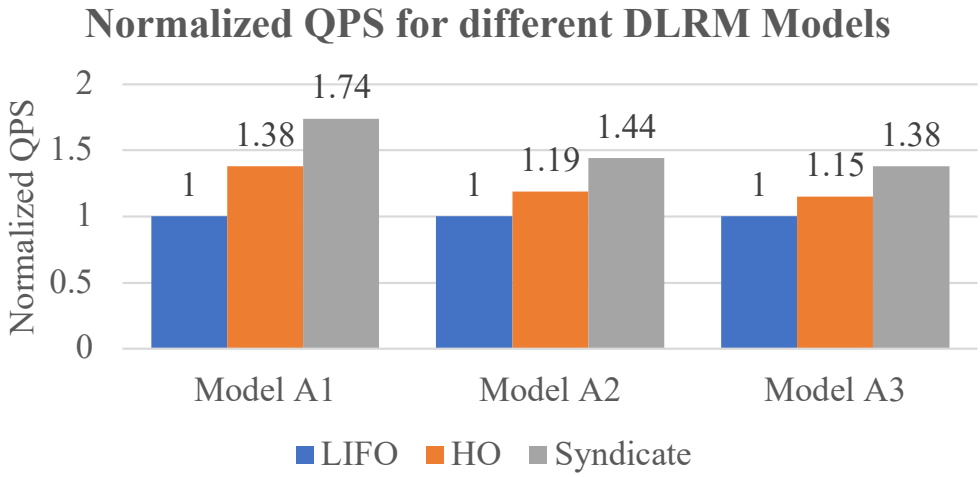


Fig. 4.11 DLRM performance for SYNDICATE compared against baselines

Figure 4.11 compares DLRM performance for SYNDICATE against the LIFO and HO baselines for all the three DLRM models. The Y-axis measures the throughput in QPS normalized against the QPS of LIFO. SYNDICATE outperforms LIFO by a factor of 1.74x, 1.44x, 1.38x for Models A1, A2, A3, respectively. SYNDICATE outperforms HO baseline by a factor of 1.26x, 1.21x, 1.2x for Models A1, A2, A3, respectively.

4.6.5.1 Sources of Improvement

Compute Idling: Figure 4.12 compares the compute idling metric for SYNDICATE against

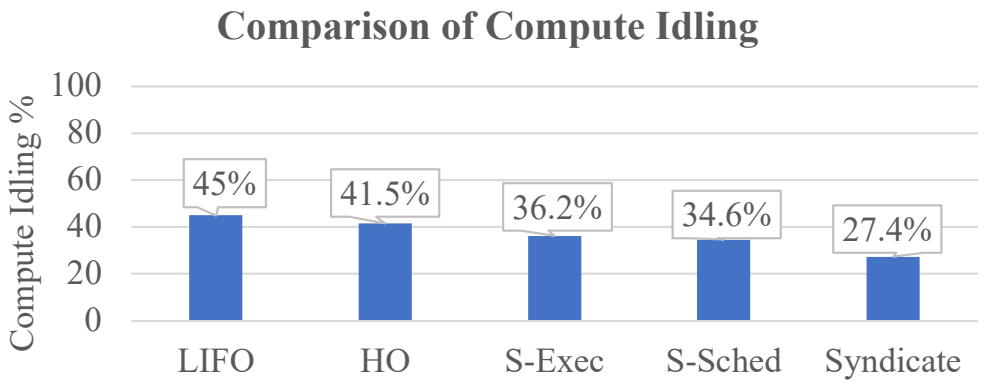


Fig. 4.12 Comparison of Compute Idling

baselines for Model A2. We observe that compute idling with SYNDICATE is 27.4% and is 1.64x, 1.51x, 1.32x, 1.26x less than the LIFO, HO, S-Sched, S-Exec baselines, respectively. This shows that SYNDICATE is better at overcoming communication bottlenecks and achieves higher overlap of compute and communication than any other baseline. It also highlights that joint optimization is beneficial as it outperforms the S-Sched and S-Exec baselines. Next, we zoom-in on each training iteration to better understand the reasons for lesser compute idling.

Zooming in on an Iteration:

We collect traces for execution of DLRM with different systems using PyTorch Kineto [15]. We illustrate these traces. to zoom-in on the execution of compute and communication events on the GPU streams for a single training iteration for Model A2. We also show the training DAG in Figure 4.14 for reference. We explain these traces one system at a time.

LIFO: LIFO prioritizes the execution of the most recently submitted collective. For reference, Figure 4.14 shows the order of submission of collectives by DLRM PyTorch trainer. To achieve LIFO, collectives need to be segmented at the right boundaries and as explained before, we use a segment oracle to do so. LIFO is the worst-performing baseline. LIFO prioritizes execution of a2a-emb-bwd over ar-top-mlp[2], which is beneficial. However, to its detriment, it also prioritizes execution of ar-bottom-mlp over a2a-emb-bwd despite a2a-emb-bwd being on the critical path. Delaying a2a-emb-bwd also delays bwd-emb compute, which delays a2a-data-prep.

HO: To amend the drawbacks of LIFO ordering, we hand optimize the scheduling order to prioritize the execution of a2a-emb-bwd as well as a2a-data-prep before ar-bottom-mlp. We

[2]We use a2a and ar as short hand for all-to-all and all-reduce, respectively.

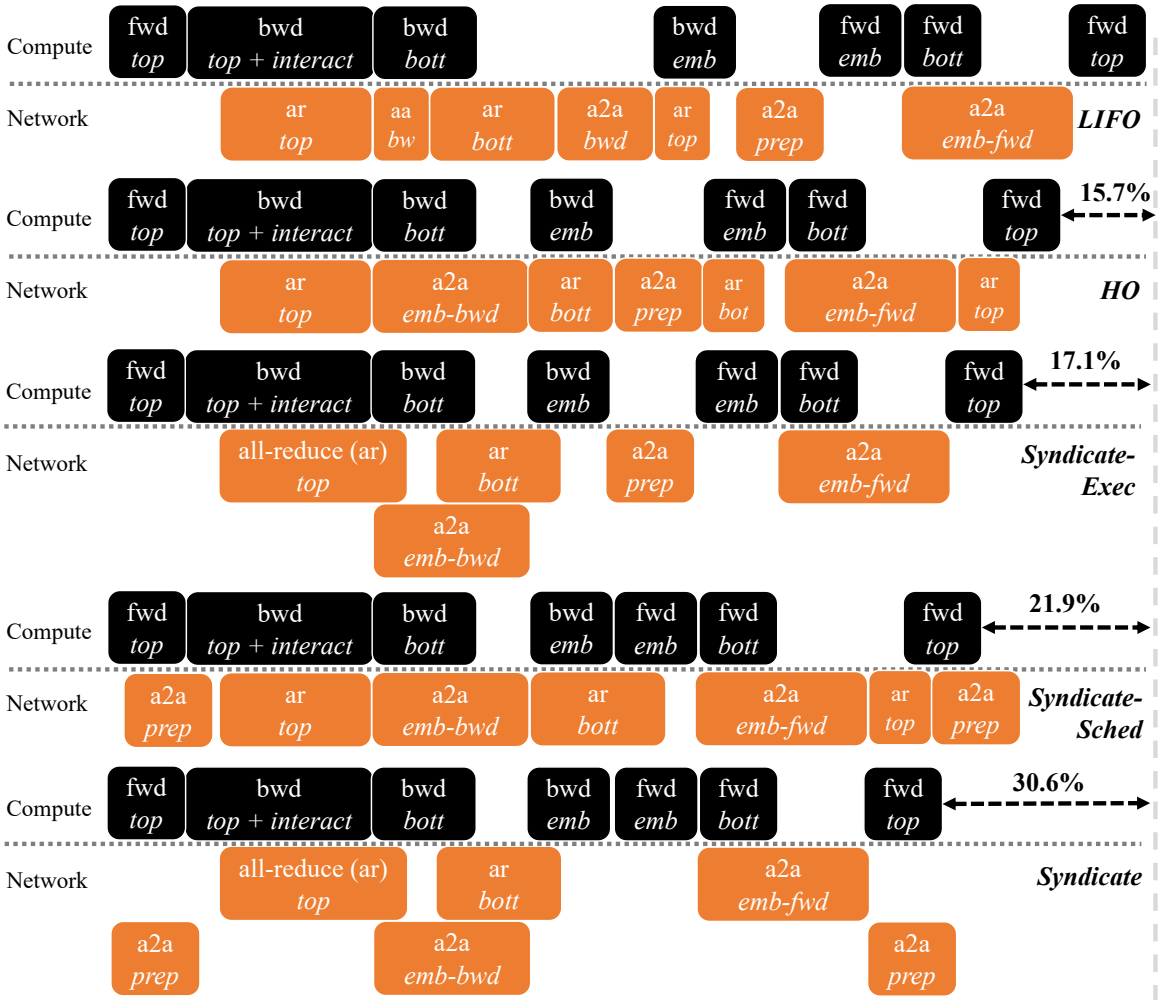


Fig. 4.13 Zooming in on every DLRM iterations for different systems

also add support for greedy execution planning for a2a collective (ar greedy optimization is available out-of-the-box). We observe that HO improves the iteration time by 15.7% as compared to LIFO. We observe that the key reason for this improvement is that HO unblocks bwd-emb and fwd-emb compute sooner and enables better overlapping of ar-bottom-mlp with these compute blocks.

S-Exec: With S-Exec, we observe that the iteration time is further improved and is 17.1% better as compared to LIFO. We observe that despite placing limiting constraints on scheduling (default FIFO scheduling), the joint optimizer in SYNDICATE finds an execution plan that assigns two different communication channels to a2a and ar and enables better communication-communication overlap by leveraging heterogeneity in the network. The ar's use a communication channel over NVLink (for intra-node) and GPUDirect RDMA (for inter-node). The a2a's use a non-intersecting communication channel over PCIe (for intra-node) and TCP/IP over Ethernet (for inter-node). Such communication-communication overlap is not possible

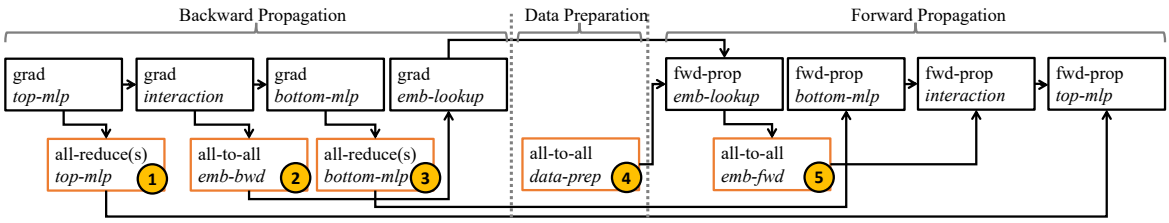


Fig. 4.14 DLRM Training DAG. The numbers represent the order in which the PyTorch modules (`nn.DistributedDataParallel` and `nn.EmbeddingBag`) submit these collectives.

with HO and LIFO as they use traditional communication stack and interfaces therein only permit one-at-a-time execution of comm-ops with greedy execution plan.

S-Sched: With S-Sched, we observe that iteration time is further improved and is 21.9% faster than LIFO, despite the constraints on execution planning (we also handicap S-Sched with choosing the default execution plan option, which is sub-optimal, for a2a). The primary reason for the improvement is that SYNDICATE's scheduler finds a superior comm-op scheduling order and SYNDICATE's enforcer enables enforcing of this order. SYNDICATE's scheduling order moves a2a-data-prep from the i^{th} iteration and moves it back in time as to overlap it with the fwd-top-mlp and bwd-top-mlp compute blocks in the $(i-1)^{th}$ iteration. The enforcer design enables this ordering due to the presence of busy loop in the `enforceOrder` procedure in Pseudocode 5. The enforcer blocks execution of all the comm-ops in the very first training iteration until a2a-data-prep collective for the next batch is submitted by the application layer. This increases the training iteration time only for the first iteration but significantly improves the training iteration time for all the subsequent iterations by unlocking pipelining.

SYNDICATE*:* With SYNDICATE, we observe that iteration time is faster than all the baselines and is 30.6% faster than LIFO. We observe that as compared to S-Sched, SYNDICATE is able to hide the overheads of ar-top-mlp by completely overlapping it with compute. SYNDICATE enables this by leveraging network heterogeneity and enabling communication-communication overlap of ar-top-mlp and a2a-emb-bwd. S-Sched, due to its execution planning constraints is unable to do so and in its scheduling order has to partially push ar-top-mlp to the very end where it cannot be overlapped with compute. In this way, SYNDICATE's joint optimizer maximizes compute-communication overlap by leveraging the benefits of communication-communication overlap.

SYNDICATE's Optimizer Plan for DLRM: Here, we summarize the key highlights of the optimizer plan that SYNDICATE finds for DLRM Model A2. In our study, we find that these observations also hold for Model A1 and Model A3.

Data Prefetch The scheduling order proposed by the optimizer moves a2a-data-prep back in time from the i^{th} iteration to the $(i-1)^{th}$ iteration. As mentioned before, this is made possible by SYNDICATE's enforcer.
a2a-ar Overlap The execution plan proposed by the optimizer overlaps all-to-all collective with all-reduce collective over two separate communication channels as explained before. SYNDICATE binds both the a2a's to the 4-way splined execution plan, the ar-bottom-mlp to the ring all-reduce execution plan and the ar-top-mlp to the tree all-reduce execution plan. This maximizes multipath network utilization and also enables greater communication-compute overlap.

4.6.6 Microbenchmarks

We use the communication microbenchmark, PARAM [13] to systematically understand the space of execution plans for different collectives to better understand the choices made by SYNDICATE's optimizer plan. SYNDICATE uses these microbenchmarks as a cost model for its joint optimizer.
Execution planning options for all-reduce: Figure 4.15 shows the effect of different all-

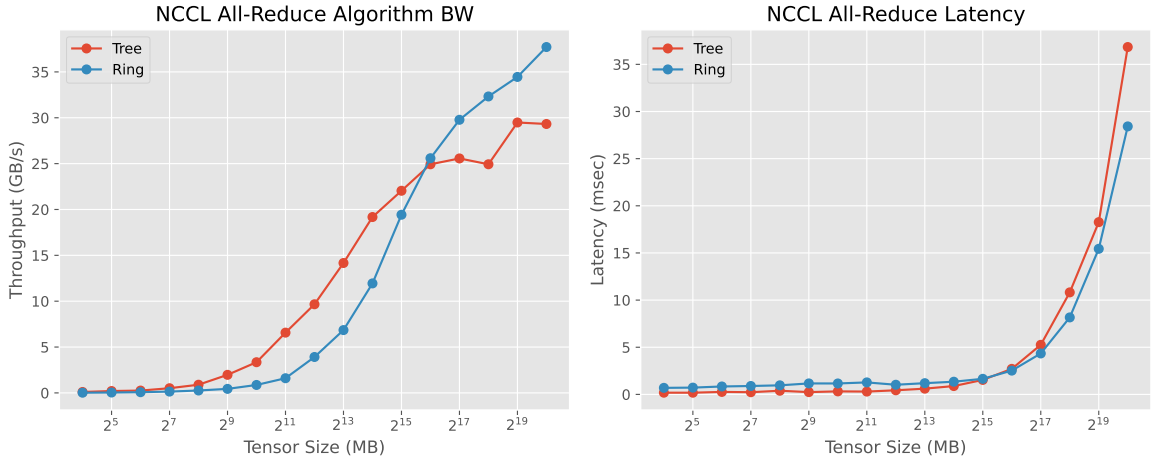


Fig. 4.15 Effect of different execution plans on all-reduce performance

reduce execution planning options in our testbed. We see that the optimal execution plan depends on the input message size. The optimal plan at small message sizes is tree all-reduce motif whereas the optimal plan at large message sizes is ring all-reduce motif. Bottom MLP is wider and induces larger (O(100's of MB) vs. top MLPs O(MB)) all-reduce collectives and explains choice of ring all-reduce and tree all-reduce for ar-bottom-mlp and ar-top-mlp, respectively.
Execution planning options for all-to-all: Figure 4.16 shows the effect of different all-to-all

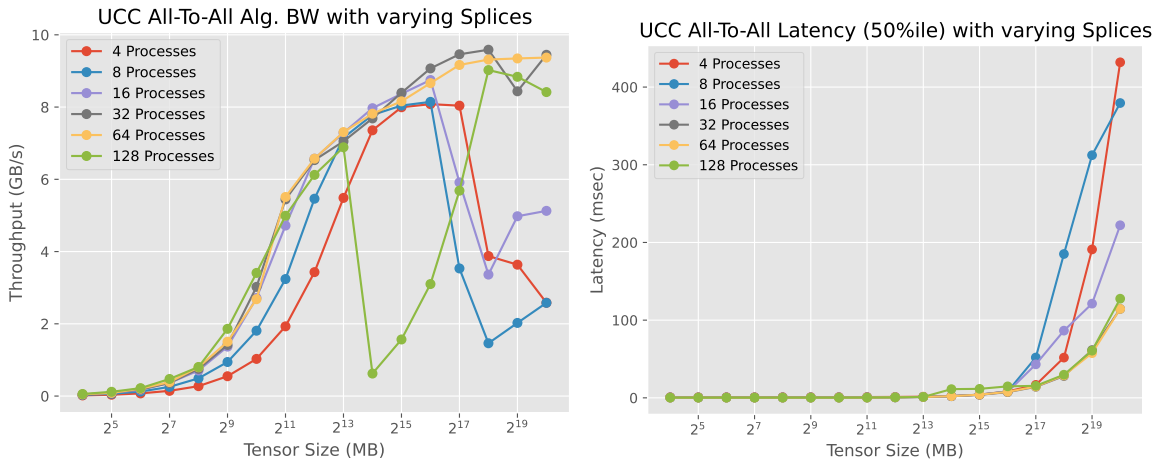


Fig. 4.16 Effect of different execution plans on all-to-all performance

execution planning options. We note that the optimal plan at small message sizes is the 1-way splined motif (simultaneous transfers to 128 destination processes from all source processes), at intermediate message sizes is the 2-way splined motif (64 Processes), and at large message sizes is the 4-way splined motif (32 Processes). The effects of incast are significant as we increase the message sizes and more splining helps reduce incast. DLRM training induces large a2a's (O(GB) message size) and SYNDICATE chooses the 4-way splined execution plan.

Execution planning options for overlap of all-reduce and all-to-all: Figure 4.17 shows

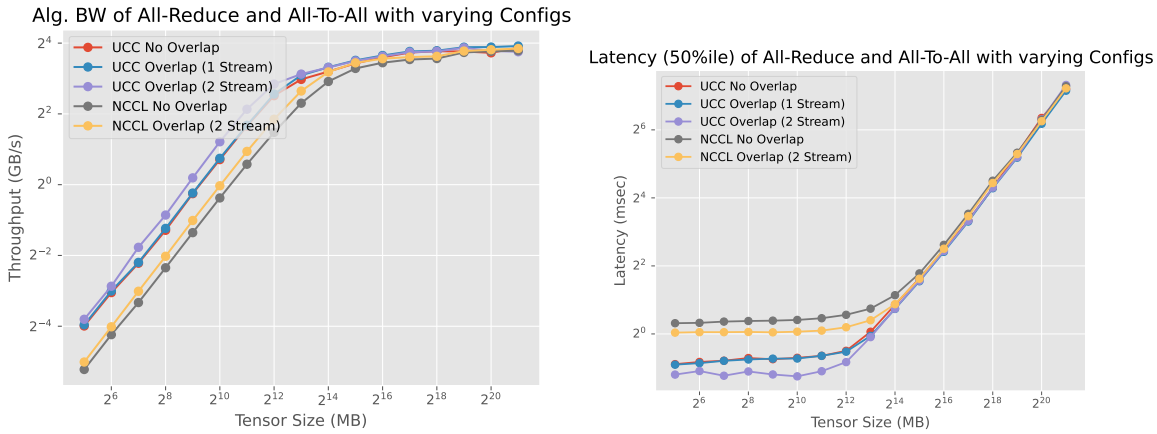


Fig. 4.17 Effect of different execution plans for all-to-all and all-reduce overlap

that the optimal execution plan is the one where all-reduce uses NCCL implementation and all-to-all use UCC implementation over non-overlapping communication channels (i.e., 2 streams). With this implementation all-to-all is CPU-driven and uses the PCIe complex and TCP/IP over Ethernet, while all-reduce is GPU-driven and uses NVLink complex and

GPUDirect RDMA. This choice is optimal (as opposed to vice versa) as all-reduce also does compute (gradient aggregation) which is faster with GPUs. SYNDICATE uses this execution plan for overlap of all-to-all and all-reduce.

4.7 Other Related Work

Table C.1 in the Appendix compares SYNDICATE against related systems.

Chapter 5

Conclusion

5.1 Contribution and Impact

In this thesis, we designed and evaluated three systems that make data analytics systems more efficient and fair. We instrument three different strategies and to do so reimagine the division of concerns and the interfaces in existing data analytics systems stack in the following three systems.

5.1.1 QOOP

In Chapter 2, we considered the problem of improving query performance in dynamic environments – e.g., in small private clusters, where resources vary with job arrivals and completions, and in clusters composed of spot instances, where resource availability changes due to changing prices. We showed that existing approaches are insufficient to adapt to dynamics because they use a fixed QEP throughout execution. We made the case for on-the-fly query re-planning and argued that it requires rethinking the division of labor among three key components of modern data analytics stacks: cluster scheduler, execution engine, and query planner. We propose a greedy re-planning algorithm, which offers provably competitive behavior, coupled with a simple cluster-wide scheduler that informs jobs of their current share. Our evaluation of a prototype using various workloads and resource profiles shows that our replanning approach driven by a simple scheduler matches or outperforms state-of-the-art solutions with complex schedulers and query planners.

5.1.2 Themis

In Chapter 3, we presented THEMIS, a fair scheduling framework for ML training workloads. We showed how existing fair allocation schemes are insufficient to handle long-running tasks and placement preferences of ML workloads. To address these challenges we proposed a new long term fairness objective in finish-time fairness. We then presented a two-level semi-optimistic scheduling architecture where ML apps can bid on resources offered in an auction. Our experiments show that THEMIS can improve fairness *and* efficiency compared to state of the art schedulers. We evaluated Themis with production cluster workloads at Microsoft and show 2.2x improvements in fairness and upto 250% improvements in efficiency.

5.1.3 Syndicate

In Chapter 4, we propose SYNDICATE that rethinks communication scheduling granularity and the interfaces in the communication stack for ML training to enable joint optimization of scheduling and execution planning. Using the novel notion of motifs and a split control/data plane architecture SYNDICATE achieves improvements of up to 74% for production scale DLRM training at Facebook as it better utilizes the network multipath opportunities in emerging training clusters.

5.2 Future Directions

We propose three broad strategies to co-optimize for fairness and efficiency: (1) top-heavy strategy with feedback loop from scheduler to execution planner, (2) bottom-heavy strategy with additional information from the execution planner to the scheduler, (3) monolithic strategy with merging of execution planner and scheduler.

Algorithmically, the monolithic strategy is most optimal as it is not constrained by the existing layering of concerns. However, it is more disruptive as it requires substantial rethinking of and reengineering of system stacks. The bottom-heavy and top-heavy strategies are less disruptive to the layering of concerns and only require incremental changes to the interfaces, however they are farther from optimal than the monolithic strategy.

In the future, there are other dynamic strategies that can be explored.

Multi-Modal Strategy: This can be achieved by making the interfaces generic enough to accomodate switching and exercising a subset of the interfaces to activate either the top-heavy or the bottom-heavy strategy as the workload characteristics or cluster resource characteristics change. In this regards, the data analytics system is multi-modal and appropriately shifts

to a different mode for different contexts. This can bridge the optimality gap to monolithic strategy.

Primal-Dual Strategy: This can be achieved by closing the loop in either the top-heavy or the bottom-heavy strategy. To get to the primal strategy, the top-heavy strategy can be further extended to send information about the most recent choice of the execution plan to the scheduler to further influence the resource allocation decision with time. This change in scheduling decision triggers the top-heavy strategy. To get to the dual strategy, the bottom-heavy strategy can be further extended to send changes in resource allocation decisions from the scheduler to the execution planner to influence execution planning decsions. This change in execution plan triggers the bottom-heavy strategy to send updated information about the execution plans to the scheduler to affect scheduling decisions. As can be seen, the primal-dual strategy lead to the same outcome, further close the gap to optimality to monolithic strategy, and blends both the bottom-heavy and top-heavy strategies while retaining the existing layering of concerns.

Instance Optimal Strategy: In all the three systems, we employ algorithms or heuristics with good worst-case guarantees and tune knobs that work well in the average case scenario. For example, `QOOP` proposes a greedy QEP switching policy to deal with resource volatility and show that its performance regression is upper bounded even in the worst-case resource volatility. In `Themis`, we propose a filtering knob to strike the optimal the trade-off between fairness and efficiency in the average case of all the workloads that we consider. However, there is room to further tailor these algorithms, heuristics and knobs on an instance-by-instance basis. Each instance can be characterized by the the distribution of future resource volatility or QEP choices in the case of `QOOP` or the distribution of job-level details and GPU cluster details in the ML training workload in the case of `Themis`. These systems can adapt on an instance-by-instance basis, whereby, instead of binding to fixed and static algorithms, heuristics and knob values, these can be changed on-the-fly. One of several ways to do this, is to train a model that takes as input the details of the instance and outputs the heuristic decisions or knob values that are most suitable for this instance.

References

[1] Ai and compute. https://openai.com/blog/ai-and-compute. Accessed: 2021-08-26.

[2] Amazon EC2. http://aws.amazon.com/ec2.

[3] Amazon Simple Storage Service. http://aws.amazon.com/s3.

[4] Apache Calcite. http://calcite.apache.org/.

[5] Apache Hadoop. http://hadoop.apache.org.

[6] Apache Hive. http://hive.apache.org.

[7] Apache Mesos 2016 Survey Report Highlights. https://goo.gl/R6a1z2.

[8] Apache Tez. http://tez.apache.org.

[9] Google Cluster Traces. https://github.com/google/cluster-data.

[10] Hadoop Private Cluster Size Statistics. https://wiki.apache.org/hadoop/PoweredBy.

[11] Microsoft Azure. http://azure.microsoft.com.

[12] Nvidia collective communication library: Optimized primitives for collective multi-gpu communication. https://github.com/NVIDIA/nccl. Accessed: Friday 1st October, 2021.

[13] Parametrized recommendation and ai model benchmark. https://github.com/facebookresearch/param. Accessed: Friday 1st October, 2021.

[14] Presto. https://prestodb.io.

[15] Pytorch kineto. https://github.com/pytorch/kineto. Accessed: 2021-08-26.

[16] Pytorch process group third-party plugin for ucc. https://github.com/facebookresearch/torch_ucc. Accessed: Friday 1st October, 2021.

[17] Storm: Distributed and fault-tolerant realtime computation. http://storm-project.net.

[18] TPC Benchmark DS (TPC-DS). http://www.tpc.org/tpcds.

[19] Trident: Stateful stream processing on Storm. http://storm.apache.org/documentation/Trident-tutorial.html.

[20] Unified collective communication (ucc). https://ucfconsortium.org/projects/ucc/. Accessed: Friday 1st October, 2021.

[21] YARN Fair Scheduler. http://goo.gl/w5edEQ.

[22] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[23] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at Internet scale. *VLDB*, 2013.

[24] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[25] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.

[26] Apache Hadoop Submarine. https://hadoop.apache.org/submarine/, 2019.

[27] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[28] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. D. Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1), 2015.

[29] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *SoCC*, 2013.

[30] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[31] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[32] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive datasets. In *VLDB*, 2008.

[33] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[34] M. Cho, U. Finkler, D. Kung, and H. Hunter. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. In *Proceedings of the 2nd SysML Conference*, 2019.

[35] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[36] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[37] C.-H. Chu, P. Kousha, A. A. Awan, K. S. Khorassani, H. Subramoni, and D. K. Panda. NV-Group: Link-Efficient Reduction for Distributed Deep Learning on Modern Dense GPU Systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, 2020.

[38] R. Cole, V. Gkatzelis, and G. Goel. Mechanism design for fair division: allocating divisible items without payments. In *Proceedings of the fourteenth ACM conference on Electronic commerce*, pages 251–268. ACM, 2013.

[39] Common Voice Dataset. https://voice.mozilla.org/.

[40] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[42] A. Desai, L. Chou, and A. Shrivastava. Random Offset Block Embedding Array (ROBE) for CriteoTB Benchmark MLPerf DLRM Model : 1000× Compression and 2.7× Faster Inference, 2021.

[43] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[44] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[45] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[46] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[47] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. *SIGCOMM*, 2012.

[48] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[49] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, 2013.

[50] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *ICDE*, 2011.

[51] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *KDD*, 2017.

[52] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[53] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[54] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[55] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[56] 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.

[57] Gurobi Optimization. http://www.gurobi.com/.

[58] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[59] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. In *SysML*, 2019.

[60] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[61] 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.

[62] J. Hellerstein. Query optimization. In P. Bailis, J. M. Hellerstein, and M. Stonebraker, editors, *Readings in Database Systems*, chapter 7. 2017.

[63] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[64] B. Huang and J. Yang. CÜmÜlÖn-d: Data analytics in a dynamic spot market. *Proc. VLDB Endow.*, 10(8):865–876, Apr. 2017.

[65] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[66] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[67] J. M. Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[68] R. Jain, D.-M. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report DEC-TR-301, Digital Equipment Corporation, 1984.

[69] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko. Priority-based parameter propagation for distributed dnn training. *arXiv preprint arXiv:1905.03960*, 2019.

[70] 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 ATC*, 2019.

[71] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. *SysML 2019*, 2019.

[72] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 463–479, 2020.

[73] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *INFOCOM*, 2012.

[74] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A distributed machine-learning system. In *CIDR*, 2013.

[75] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[76] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[77] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[78] H. Liu. Cutting MapReduce cost with spot market. In *HotCloud*, 2011.

[79] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, 2010.

[80] K. Mahajan, M. Chowdhury, A. Akella, and S. Chawla. Dynamic query re-planning using {QOOP}. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 253–267, 2018.

[81] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[82] M. Marcus, B. Santorini, and M. A. Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. 1993.

[83] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[84] Message Passing Interface Forum. http://www.mpi-forum.org/. Accessed: Friday 1st October, 2021.

[85] Microsoft Philly Trace. https://github.com/msr-fiddle/philly-traces, 2019.

[86] D. Mudigere, Y. Hao, J. Huang, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. R. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao. High-performance, distributed training of large-scale deep learning recommendation models. *CoRR*, abs/2104.05158, 2021.

[87] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[88] 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.

[89] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[90] NVIDIA. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect, 2011. Accessed: Friday 1st October, 2021.

[91] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[92] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[93] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, page 3. ACM, 2018.

[94] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[95] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, V. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *SIGCOMM*, 2015.

[96] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2020.

[97] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[98] J. Rasley, Y. He, F. Yan, O. Ruwase, and R. Fonseca. Hyperdrive: Exploring hyperparameters with pop scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 1–13. ACM, 2017.

[99] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[100] M. Seo, A. Kembhavi, A. Farhadi, and H. Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

[101] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, et al. Ucx: an open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015.

[102] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. SpotCheck: Designing a derivative IaaS cloud on the spot market. In *EuroSys*, 2015.

[103] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[104] M. Smelyanskiy. Zion: Facebook next-generation large memory training platform. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–22. IEEE Computer Society, 2019.

[105] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[106] Y. Ueno and R. Yokota. Exhaustive Study of Hierarchical AllReduce Patterns for Large Messages Between GPUs. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 430–439, May 2019.

[107] H. R. Varian. Equity, envy, and efficiency. *Journal of Economic Theory*, 9(1):63 – 91, 1974.

[108] 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.

[109] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC*, 2013.

[110] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[111] R. Viswanathan, G. Ananthanarayanan, and A. Akella. Clarinet: WAN-aware optimization for analytics queries. In *OSDI*, 2016.

[112] A. Vulimiri, C. Curino, B. Godfrey, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[113] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica. Blink: Fast and generic collectives for distributed ml. *arXiv preprint arXiv:1910.04940*, 2019.

[114] WMT16 Dataset. http://www.statmt.org/wmt16/.

[115] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. Netpilot: automating datacenter network failure mitigation. In *SIGCOMM*, 2012.

[116] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[117] 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.

[118] J. Yamagishi. English multi-speaker corpus for cstr voice cloning toolkit. *URL http://homepages. inf. ed. ac. uk/jyamagis/page3/page58/page58. html*, 2012.

[119] J. Yin, S. Gahlot, N. Laanait, K. Maheshwari, J. Morrison, S. Dash, and M. Shankar. Strategies to deploy and scale deep learning on the summit supercomputer. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, pages 84–94, 2019.

[120] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[121] 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 *EuroSys*, 2010.

[122] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[123] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

[124] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

[125] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[126] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 390–404. ACM, 2017.

[127] L. Zheng, C. Joe-Wong, C. W. Tan, M. Chiang, and X. Wang. How to bid the cloud. In *SIGCOMM*, 2015.

Appendix A

QOOP Proofs

A.1 Proofs for Section 2.4

PROOF OF THEOREM 2.1. We will fix a query and an online algorithm, and will define a sequence of resource allocations based on what the online algorithm does. The query has m possible QEPs. For $i \in [m]$, the ith QEP has requirements (a_i, b_i) with $a_i = 2^i$ and $w_i = a_i b_i = 2^{m-i}W$. Here $W > 1$ is chosen to be a large enough constant so that $w_i - a_i \approx w_i$ for all i.

The scheduler runs in m phases indexed by $j \in [m]$ with some of the phases being empty. Starting at $j = m$ down to $j = 1$, in the jth phase the scheduler gives the query $a_j = 2^j$ units of resource per time step until for some $i \leq j$, QEP i finishes $w_i - a_i$ amount of work; We say that QEP i reaches "near-completion". Observe that as soon as a QEP reaches near-completion, the scheduler starves it, and it cannot finish. At the end of the m phases, the scheduler gives the query one more unit of resource, allowing QEP 1 to finish, and the query terminates.

Let $i_1 \leq m$ denote the index of the QEP that reaches near-completion at the end of the $j = m$ phase. Observe that if $i_1 < m$, the phases $j = m-1$ to $j = i_1$ are empty, and the scheduler runs phase $i_1 - 1$ next. Let $i_2 < i_1$ denote the index of the QEP that reaches near-completion at the end of the $j = i_1 - 1$ phase. Once again, if $i_2 < i_1 - 1$, the phases $j = i_1 - 2$ to $j = i_2$ are empty, and the scheduler runs phase $i_2 - 1$ next. This continues until at the end of the tth phase for some t, QEP 1 reaches near-completion, that is, $i_t = 1$. At this point the scheduler gives the algorithm one more unit of resources, and the query terminates.

The choices of the online algorithm boil down to the sequence $m \geq i_1 > i_2 > \cdots > i_t = 1$. We will now argue that regardless of the choices for these indices, a hindsight optimal

algorithm can always complete at least two copies of the query in the same amount of time that the online algorithm takes to complete the query.

In particular, the first phase amounts to a total $w_{i_1} - a_{i_1}$ amount of work done by the online algorithm. Since the resource allocation in this phase is at least a_m per time step, the optimal algorithm can run QEP m in this phase, finishing $(w_{i_1} - a_{i_1})/w_m$ copies of the QEP. Likewise, in the first $\ell+1$ phases, the online algorithm does a total of $W_{\ell+1} = \sum_{t' \le \ell+1}(w_{i_{t'}} - a_{i_{t'}-1})$ amount of work. In the same amount of time, the optimal algorithm can run $W_{\ell+1}/w_{i_\ell - 1}$ copies of the QEP $i_\ell - 1$.

For large enough W, the competitive ratio of the algorithm is then

$$\begin{aligned}\max_{\ell \le t-1} \frac{\sum_{t' \le \ell+1} w_{i_{t'}}}{w_{i_\ell - 1}} &= \max_{\ell \le t-1} \frac{\sum_{t' \le \ell+1} 2^{m-i_{t'}}}{2^{m-i_\ell+1}} \\ &= \max_{\ell \le t-1} \frac{\sum_{t' \le \ell} 2^{m-i_{t'}} + 2^{m-i_{\ell+1}}}{2^{m-i_\ell+1}}\end{aligned}$$

Now recall that for every ℓ, $i_{\ell+1} \le i_\ell - 1$. Suppose that for some ℓ, $i_{\ell+1} < i_\ell - 1$. Then, the term in the above maximization corresponding to that ℓ is at least $2^{m-i_{\ell+1}}/2^{m-i_\ell+1} \ge 2$. On the other hand, if for every ℓ we have $i_{\ell+1} = i_\ell - 1$, then the last term in the maximization corresponding to $\ell = t-1$ is:

$$\frac{\sum_{i=1}^{i=m} 2^{m-i}}{2^{m-1}} = 2 - 2^{-(m-1)}$$

For large enough m, then, the above maximum is arbitrarily close to 2. □

PROOF OF THEOREM 2.2. We first prove a competitive ratio of 2 for the setting where for every $i > 1$ we have $w_i \le \frac{1}{2} w_{i-1}$. Fix an adversarial sequence of resource allocations. Suppose that for some $i \in [n]$, the offline optimal algorithm takes t timesteps to run two copies of the QEP i. We will show that the online algorithm completes the query within the t time steps. Consider the subset S of these t time steps when the resource allocation is at least a_i. Since the offline optimum cannot run QEP i in timesteps outside of S, the total amount of work done by the offline optimum during steps in S is at least $2w_i$.

During the timesteps in S, the online algorithm sometimes runs a QEP j with $j < i$, and other times a QEP j with $j \le i$. Call the latter subset of steps "progressive" and the former subset of steps "extra". Note that the total amount of work done by the online algorithm during the "extra" steps is at most $\sum_{j>i} w_j \le w_i$ using the fact that $w_j \le \frac{1}{2} w_{j-1}$ for all $j > 1$.

Therefore, the amount of work available for the online algorithm in the progressive steps is at least $2w_i - w_i = w_i$.

We now focus on the progressing steps. Recall again that during these steps, the online algorithm runs QEPs with resource requirement smaller than or equal to that of i. Let w^* denote the minimum work remaining in any of the QEPs j with $j \leq i$ at any point of time. Observe that during any progressive step, since the resource allocation is at least a_i, the online algorithm can run any of the QEPs j with $j \leq i$. Our online algorithm chooses to run the one with the smallest work remaining, namely w^*. Accordingly, w^* decreases linearly with the amount of work available during the progressive steps, and reaches 0. When w^* reaches 0, this means that one of the QEPs terminates and the query is completed. This completes our analysis.

We now extend this analysis to the more general setting where the online algorithm discards some QEPs in its first step. Suppose that the offline optimum runs a QEP that has not been discarded by the online algorithm, then the above analysis continues to apply, and we obtain a competitive ratio of 2. Suppose instead that the offline optimum uses one of the QEPs discarded by the online algorithm. We can then argue a slightly worse upper bound on the competitive ratio. In particular, consider such a QEP $a_i \times b_i$. Recall that we eliminated this QEP because there is another QEP $a_j \times b_j$ with $w_j < 2w_i$. If the offline optimum can run 4 copies of the QEP i within some amount of time t, then it can run 2 copies of the QEP j within the same amount of time t, and then we can carry out the above argument to show that the online algorithm finishes the query within this same amount of time t. Therefore, the competitive ratio of the online algorithm is at most 4. □

Appendix B

Themis Proofs

PROOF OF THEOREM 3.1. Examples in Figure 3.5 and Section 3.3.1.2 shows that DRF violates SI, EF, and PE. Same examples hold true for LAS policy in Tiresias. The service metric i.e. the GPU in Instance 1 and Instance 2 is the same for A1 and A2 in terms of LAS and is deemed a fair allocation over time. However, Instance 1 violates SI as A1 (VGG16) and A2 (VGG16) would prefer there own independent GPUs and Instance 2 violates EF and PE as A2 (VGG16) prefers the allocation of A1 (Inception-v3) and PE as the optimal allocation after taking into account placement preferences would interchange the allocation of A1 and A2. □

PROOF OF THEOREM 3.2. We first show that the valuation function, $\rho(.)$, for the case of ML jobs is homogeneous. This means that $\rho(.)$ has the following property: $\rho(m * \overrightarrow{G}) = m * \rho\overrightarrow{G}$.

Consider a job with GPUs spread across a set of some M machines. If we keep this set of machines the same, and increase the number of GPUs allocated on these same set of machines by a certain factor then the shared running time (T_{sh}) of this job decreases proportionally by the same factor. This is so because the slowdown, $\mathscr{S}$ remains the same. Slowdown is determined by the slowest network interconnect between the machines. The increased allocation does not change the set of machines M. The independent running time (T_{id}) remains the same. This means that ρ also proportionally changes by the same factor.

Given, homogeneous valuation functions, the PA mechanism guarantees SP, PE and EF [38]. However, PA violates SI due to the presence of hidden payments. This also make PA not work-conserving. □

PROOF OF THEOREM 3.3. With multi-round auctions we ensure truth-telling of ρ estimates in the visibility phase. This is done by the AGENT by using the cached $\rho(.)$ estimates from the last auction the app participated in. In case an app gets leftover allocations from the

leftover allocation mechanism, the AGENT updates the ρ estimate again by using the cached $\rho(.)$ table. In this way we guarantee SP with multi-round auctions.

As we saw in Theorem 3.2, an auction ensures PE and EF. In each round, we allocate all available resources using auctions. This ensures end-to-end PE and EF.

For maximizing sharing incentive, we always take a fraction $1-f$ of apps in each round. A wise choice of f ensures that we filter in all the apps with $\rho > 1$ that have poor sharing incentive. We only auction the resources to such apps which maximizes sharing incentive. □

Appendix C

Syndicate

C.0.1 Transformation Operator Algebra

We now present the algebra for the motif transformation operators. We denote the segmentation operator by $\stackrel{s}{=}$ and the splining operator by $\stackrel{p}{=}$. Note that the algebraic rules presented below are not exhaustive and are extensible. Here, we present the algebraic rules that we use in the context of DLRM to transform all-reduce and all-to-all collectives into motifs. We first go over the various symbols used in the algebra.

$\texttt{N}$: Total number of Processes

$\texttt{PG[0:N]}$: Process IDs involved in a Motif

$\texttt{T}_i\texttt{[0:D]}$: Tensor of size D on Process $\texttt{P}_i$

$\texttt{T}_i\texttt{[0:N,0:D]}$: N Tensors of size D on Process $\texttt{P}_i$ with first dimension indicating destination Process ID

$\|$: Parallel Execution

$\rightarrow$: Sequential Execution

$\stackrel{s}{=}$: Segmentation Transformation

$\stackrel{p}{=}$: Splining Transformation

$\texttt{AR}$: `all-reduce motif`

$\texttt{AA}$: `all-to-all motif`

$\texttt{RE}_r$: `reduce motif` with root Process $\texttt{P}_r$

$\texttt{RS}$: `reduce-scatter motif`

$\texttt{BC}_r$: `broadcast motif` with root Process $\texttt{P}_r$

$\texttt{AG}$: `all-gather motif`

$\texttt{COLL(T}_i\texttt{[:], PG[IDs])}$: Motif COLL with input tensor $\texttt{T}_i$ executing on each Process $\texttt{P}_i$ for all i in IDs

We now present the algebraic rules for transforming the all-reduce motif using the segment and spline operators.

Segmented All-Reduce: First, we show application of the segment operator which splits the input tensor at all the processes and converts an all-reduce motif into smaller all-reduce motifs over the splits. Each smaller all-reduce motif are independent and can execute at the same time in parallel.

$$\texttt{AR(T}_i\texttt{[0:D], PG[0:N])} \stackrel{\texttt{S}}{=} \|_{s=0}^{\frac{D}{d}-1} \texttt{AR(T}_i\texttt{[s*d:(s+1)*d], PG[0:N])}$$

Ring All-Reduce: Next, we show an instance of the spline operator that divides the pattern in original all-reduce into two sub-patterns: reduce-scatter motif followed by the all-gather motif. The reduce-scatter motif does aggregation and the all-gather motif broadcasts the aggregated result. The reduce-scatter and all-gather motifs induce a pattern of communication over a ring, where the processes are arranged in a ring and the tensor is divided into N pieces. Each process $\texttt{P}_i$ does a point-to-point transfer of the (i+r)%N piece to its neighboring process in the ring in the $\texttt{r}^{th}$ round for N rounds.

$$\begin{aligned}
\texttt{AR(T}_i\texttt{[0:D], PG[0:N])} &\stackrel{\texttt{C}}{=} \texttt{RS(T}_i\texttt{[0:D], PG[0:N])} \\
&\quad \rightarrow \texttt{AG(T}_i\texttt{[0:D], PG[0:N])} \\
\texttt{RS(T}_i\texttt{[0:D], PG[0:N])} &= \texttt{ring pattern of communication} \\
\texttt{AG(T}_i\texttt{[0:D], PG[0:N])} &= \texttt{ring pattern of communication}
\end{aligned}$$

Tree All-Reduce: Next, we show an instance of the spline operator that divides the pattern in the original all-reduce into three smaller sub-patterns: reduce motif followed by a smaller all-reduce motif followed by broadcast motif. The same spline operator algebraic can be recursively applied to the smaller all-reduce motif. Recursive application results in a hierarchical tree pattern of communication where several reduce motifs first aggregate results in a tree like fashion at a single root process and several broadcast motifs broadcast the aggregated result from the root process in a tree like fashion until it is updated at all the processes. Each reduce motif results in a convergent pattern of communication where all the processes involved in the reduce send their tensors to the root process where it is aggregated. Each broadcast motif results in a divergent pattern of communication where the root process sends its tensor to all the processes involved in the broadcast motif.

$$\begin{aligned}
\texttt{AR(T}_i\texttt{[0:D], PG[0:N])} &\stackrel{\texttt{C}}{=} \|_{c=0}^{\frac{N}{n}-1} \texttt{RE}_{c*n}\texttt{(T}_i\texttt{[0:D], PG[c*n:(c+1)*n])} \\
&\quad \rightarrow \texttt{AR(T}_i\texttt{[0:D], PG[}\cup_{c=0}^{\frac{N}{n}-1} \texttt{c*n])} \\
&\quad \rightarrow \|_{c=0}^{\frac{N}{n}-1} \texttt{BC}_{c*n}\texttt{(T}_i\texttt{[0:D], PG[c*n:(c+1)*n])} \\
\texttt{RE}_j\texttt{(T}_i\texttt{[0:D], PG[j:j+n])} &= \texttt{convergent pattern of communication} \\
\texttt{BC}_j\texttt{(T}_i\texttt{[0:D], PG[j:j+n])} &= \texttt{divergent pattern of communication} \\
\texttt{AR(T}_i\texttt{[0:D], PG[}\cup_{c=0}^{\frac{N}{n}-1} \texttt{c*n])} &= \text{recursive application of } \stackrel{\texttt{C}}{=} \text{ induces} \\
&\quad \texttt{tree pattern of communication}
\end{aligned}$$

Segmented and Splined All-To-All: Next, we show examples of segmenting and splining an all-to-all collective into smaller motifs. With segmentation, the tensor at all the processes is split and the original all-to-all is deconstructed into several smaller all-to-all motifs over the split tensors. With splining, the pattern of communication in the original all-to-all motif with a clique of point-to-point transfers between all the processes is broken down into smaller all-to-all motifs with smaller patterns where each process $\texttt{P}_i$ initiates point-to-point transfers to a subset of destination processes (with ids in the range `(i+c*n)%N:(i+(c+1)*n)%N`). Here, `n` parameterizes the all-to-all splining operator with larger `n` resulting in breaking the original all-to-all into fewer all-to-all motifs with larger sub-patterns.

$$\texttt{AA(T}_i\texttt{[0:N,0:D], PG[0:N])} \overset{\text{S}}{=} \|_{s=0}^{\frac{D}{d}-1} \texttt{AA(T}_i\texttt{[0:N, s*d:(s+1)*d], PG[0:N])}$$

$$\texttt{AA(T}_i\texttt{[0:N,0:D], PG[0:N])} \overset{\text{C}}{=} \|_{c=0}^{\frac{N}{n}-1} \texttt{AA(T}_i\texttt{[(i+c*n)\%N:(i+(c+1)*n)\%N, 0:D], PG[0:N])}$$

		TicTac [59]	P3 [69]	Blink [113]	ByteScheduler [94]	Syndicate
Execution	N/W Tput.	×	×	✓	✓	✓
	N/W Topo.	×	×	✓	×	✓
	N/W Ops	Push-Pull	Push-Pull	All-Reduce	Push-Pull; All-Reduce	Push-Pull; Collectives
Scheduling	Preemptible	✓	✓	—	✓	✓
	Models	DAGs	Layer-by-Layer	—	Layer-by-Layer	General DAGs
	Frameworks	PS	PS	—	PS; ∼ P2P	PS; P2P
	Policy	DAG driven	LIFO	—	LIFO	DAG Optimal
Joint Optimization		×	×	—	×	✓

Table C.1 Comparison of systems optimizing communication operations for training workloads

C.0.2 Other Related Work

Several works minimize communication overheads with the aim of speeding-up training. These optimize two main aspects of communication operations: scheduling and execution. Scheduling look at reordering communication operations to maximize overlap of compute and communication. Execution optimizations look at speeding up individual communication operations. These optimization use batching to improve link utilization, enable mutipath in collectives to make better use of heterogeneous links in the network, and enable preemption to enable scheduling optimizations. Existing works do not jointly optimize both these concerns. Table C.1 shows how SYNDICATE dominates these systems.