Efficient Resource Management for Deep Learning Clusters

by

Juncheng Gu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2021

Juncheng Gu

jcgu@umich.edu

ORCID iD: 0000-0002-3315-5784

To my wife and my parents for their unwavering love and support.

ACKNOWLEDGMENTS

PhD study is a long journey. I can not imagine how I would complete it without the help and support of my supervisors, family, colleagues, and friends.

First and foremost, I want to give special thanks to my supervisors, Professor Kang G. Shin, and Professor Mosharaf Chowdhury, for their invaluable advice, continuous support, and extraordinary patience during my PhD study. Professor Kang G. Shin grants me the freedom to explore research topics I like, and always offers me insightful guidance at the right moments. Professor Mosharaf Chowdhury instrumentally teaches me how to approach research in the right manner, and warmly encourages me when I am facing hardships. I am truly fortunate to have the two great supervisors who share their vision and wisdom in these years.

I am thankful to Professor Baris Kasikci, Harsha V. Madhyastha, and Ying Lei for serving on my thesis committee. Their valuable and constructive comments and suggestions help me improve the quality of my thesis.

I would like to thank my collaborators, Youngmoon Lee, Yiwen Zhang, and Haizhong Zheng. I will never forget the days and nights when we worked together for experiments, brainstorm, and discussion. I am grateful to Xiaoen Ju for his guidance during the first year of my PhD study.

It is a great fortune to be a member of both the Real-Time Computing Laboratory (RTCL) and the Symbiotic Lab where I learned a lot and received lots of help. I would like to thank all RTCL members: Chun-Yu Chen, Dongyao Chen, Kyoung Tak Cho, Kassem Fawaz, Huan Feng, Arun Ganesan, Liang He, Suining He, Taeju Park, Mert Pese, Yu-Chih

Tung, Xiufeng Xie; and Symbiotic members: Fan Lai, Hasan Al Maruf, Peifeng Yu, Jie You. I also want to thank my friends in Ann Arbor, especially the family of Zhe Wu.

Finally and most importantly, my sincere gratitude goes to my family for their deep love and tremendous support. My father, Xiaohong Gu, and my mother, Biyu Zhang, have persistently given their unconditional care and support throughout my life. My wife, Yujie Yan, is the light of my life that is always with me and encourages me. Her love has been supporting me to go through the darkness. My son, Yijian Gu, is the best gift I received during my PhD journey.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

MLFQ	Multi Level Feedback Queue	79
SERPT	Shortest Expected Remaining Processing Time	104
SLO	Service Level Objective	148

ABSTRACT

Deep Learning (DL) is gaining rapid popularity in various domains, such as computer vision, speech recognition, etc. With the increasing demands, large clusters have been built to develop DL models (i.e., data preparation and model training). DL jobs have some unique features ranging from their hardware requirements to execution patterns. However, the resource management techniques applied in existing DL clusters have not yet been adapted to those new features, which leads to resource inefficiency and hurts the performance of DL jobs.

We observed three major challenges brought by DL jobs. First, data preparation jobs, which prepare training datasets from a large volume of raw data, are memory intensive. DL clusters often over-allocate memory resource to those jobs for protecting their performance, which causes memory underutilization in DL clusters. Second, the execution time of a DL training job is often unknown before job completion. Without such information, existing cluster schedulers are unable to minimize the average Job Completion Time (JCT) of those jobs. Third, model aggregations in Distributed Deep Learning (DDL) training are often assigned with a fixed group of CPUs. However, a large portion of those CPUs are wasted because the bursty model aggregations can not saturate them all the time.

In this thesis, we propose a suite of techniques to eliminate the mismatches between DL jobs and resource management in DL clusters. First, we bring the idea of memory disaggregation to enhance the memory utilization of DL clusters. The unused memory in data preparation jobs is exposed as remote memory to other machines that are running out of local memory. Second, we design a two-dimensional attained-service-based scheduler to optimize the average JCT of DL training jobs. This scheduler takes the temporal and spatial characteristics of DL training jobs into consideration and can efficiently schedule them without knowing their execution time. Third, we define a shared model aggregation service to reduce the CPU cost of DDL training. Using this service, model aggregations from different DDL training jobs are carefully packed together and use the same group of CPUs in a time-sharing manner. With these techniques, we demonstrate that huge improvements in resource efficiency and job performance can be obtained when the cluster's resource management matches with the features of DL jobs.

CHAPTER I

Introduction

In recent years, DL has gained significant successes in many research domains, such as image classification [93], speech recognition [95], and machine translation [68]. More importantly, it has become the driving power in a variety of application disciplines, such as robotics [104], autonomous driving [55, 139], and healthcare [88], and has remarkably changed our daily lives.

The core of a DL application is to build and train a particular mathematical model (i.e., deep neural network). The model contains many parameters that are tuned throughout the training process to fit the observed data closely. After training, the model can predict outcomes for the new data based on the data patterns it learned.

With the popularity of DL applications, an increasing number of DL models are being developed. A $10.5\times$ increase of DL jobs has been observed in Microsoft from 2016 to 2018 [99]. Meanwhile, more and more large-scale DL clusters have been built and shared among many users to meet the rising demands [41, 3, 4, 10]. Training a DL model is typically compute-intensive and heavily reliant on powerful GPU for acceleration. Therefore, different from the traditional clusters, DL clusters are often equipped with a large number of GPU devices.

Despite the rapid innovations in hardware (e.g., GPU, TPU [101]), the software systems for resource management in DL clusters are lagging behind. Existing DL clusters still apply the resource management techniques that were designed for traditional big-data clusters [164, 99]. However, those techniques have not yet been adapted to the new features of DL applications, which leads to resource inefficiency of clusters and hurts the performance of DL jobs as well. For example, existing cluster managers perform poorly in scheduling DL training jobs (Chapter III). The unpredictable execution time of DL training jobs makes them unable to optimize the overall JCT.

This thesis focuses on the mismatches between DL jobs and resource management techniques in existing DL clusters. We propose a suite of techniques to mitigate them with the objectives of improving resource efficiency and the performance of DL jobs.

In the remainder of this chapter, we introduce some background information of DL jobs, the major challenges in DL clusters, and summarize the contributions of this thesis.

1.1 Background: Developing Deep Learning Models

This section introduces the background knowledge of DL models. Generally, developing a DL model contains two major steps: data preparation and model training (Figure 1.1).

1.1.1 Data Preparation

A DL model contains the patterns learned from a large amount of observed data. Therefore, one important foundation of developing DL models is to have well-prepared training datasets. For example, the success of ImageNet [67] dataset has greatly driven the evolu-

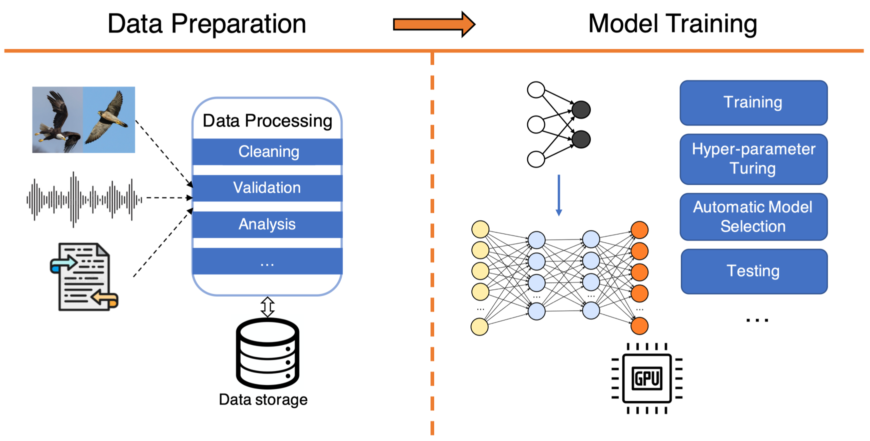
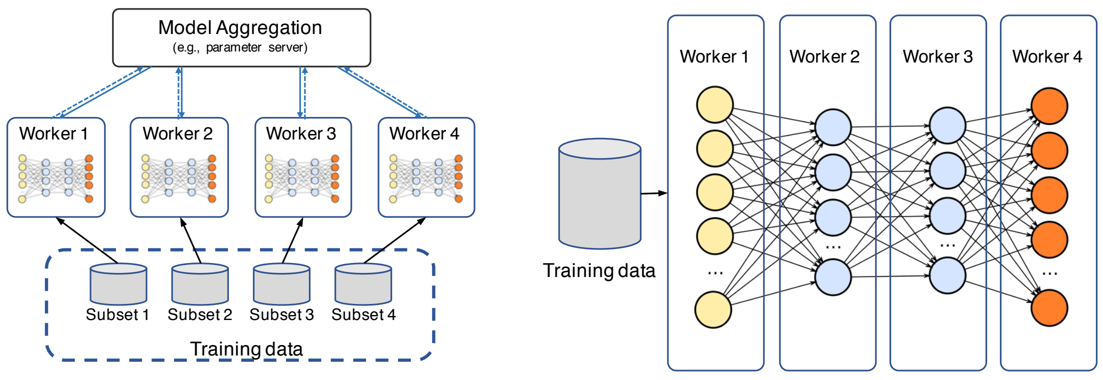


Figure 1.1: Developing DL models. There are two steps: data preparation and model training.

tion in image classification. In the data preparation step, a large volume of raw data are processed and converted into usable datasets. The processing operations include but are not limited to data cleaning, validation, and labeling. There is no existing end-to-end platform that automatically runs data preparation for DL yet. Snorkel [144] is the first tool that can automatically label the training data. Normally, developers have to customize data preparation systems by utilizing the existing data processing frameworks (e.g., Spark [32]).

1.1.2 Model Training

Model training is the process that uses a learning algorithm to adjust model parameters to make the DL model fit the training data most closely. DL training works in an iterative fashion. In each iteration, the training process operates on a few samples of training data called mini-batch. It first performs a *forward pass* with one mini-batch of input data. The error between the outputs of the *forward pass* and the desired outputs will be propagated over the model through a *backward pass* where the gradient for each model parameter is generated. Before the end of the iteration, the model parameters will be updated using



(a) Data Parallelism (b) Model Parallelism

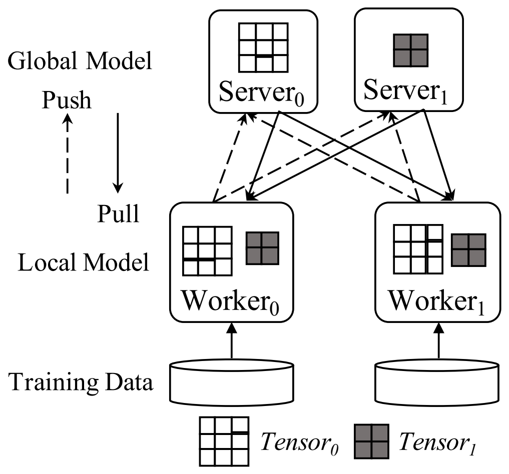
Figure 1.2: Two major patterns of DDL training. Data parallelism in (a) uses Parameter Server (PS) for model aggregation.

their gradients by the model updating function. Due to the very large number of model parameters in a single model, the *forward-and-backward* computation is often executed by GPUs for acceleration. Typically, millions of such iterations have to be executed before the model reaches its accuracy target.

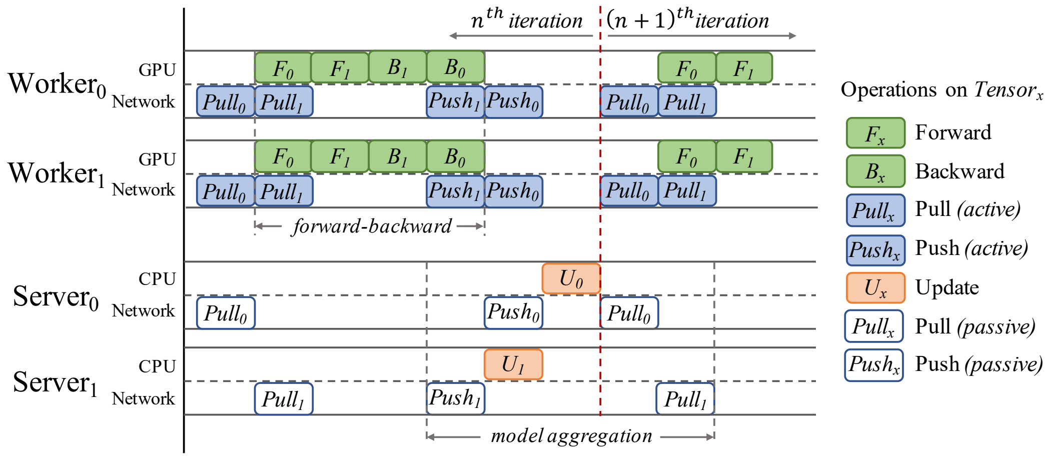
1.1.2.1 Distributed Training

To deal with the rapid increase of training datasets and model complexity, DDL training, which leverages multiple (GPU) workers to work on the same model in parallel, is becoming more prevalent [140, 164]. There are two major approaches of distributed training: data parallelism and model parallelism, both of which are available in popular DL frameworks [35, 25, 31, 20].

Data Parallelism. The most common choice for DDL training is data parallelism. Each worker node operates on its local copy of the model (i.e., parameters); the training dataset is divided into equal-sized subsets to feed those workers (Figure 1.2a). In each iteration, workers perform *forward-and-backward* computation and generate their local gradients in-



(a) Job components.



(b) Execution timeline.

Figure 1.3: A toy example of data-parallel distributed training. Parameter server (PS) is applied for model aggregation. There are 2 workers and 2 server instances in this job. The DL model has 2 tensors. Each server instance is in charge of one tensor and handles the related aggregation operations. (a) presents the key job components in this example. (b) shows the execution timeline of all participants(i.e., workers and servers) on different resource (e.g., GPU, CPU, and network). After the n^{th} iteration, the job proceeds to the $(n+1)^{th}$ iteration with the same operations. For simplicity, the width of each operation block is equalized and does not represent the actual execution time.

dependently. Different from single-worker training, *data-parallel* distributed training needs to execute *model aggregation*[1] before starting the next training iteration (Figure 1.3b). In model aggregation, the local gradients from distributed workers are *aggregated* together and transformed into global gradients for model updating. After updating the model pa-

[1]Also known as tensor aggregation and gradient aggregation.

rameters, the new model parameters are pulled back by each worker for the next training iteration. Parameter server (PS) [112, 141] (Figure 1.3a) is the most common implementation of *model aggregation* in popular *DL* frameworks. The *server* instance in PS is independent of the training workers. It hosts the most updated copy (i.e., master copy) of model tensors ((i.e., a group of parameters) and updates them using the *aggregated* results (Figure 1.3b).

Model Parallelism. Instead of dividing training data, model parallelism partitions model parameters across worker nodes (Figure 1.2b), which makes it possible to train a large model whose size exceeds the capacity of a single worker. Each worker in the training job operates on a subset of model parameters on the same training data. Data communication happens when passing intermediate results between workers who hold the neighbor parameters. However, implementing model parallelism, which needs to partition the model, requires full knowledge of the training task. Therefore, model parallelism is less popular than data parallelism.

Beyond those two types of distributed training, several alternatives have been proposed with different optimization objectives. Tofu [162] enables training extra-large DL models by partitioning the dataflow graph of model tensors, which has less GPU memory footprint than vanilla model parallelism. PipeDream [130] proposes *pipeline parallelism* that achieves faster DL training by combing data and model parallelism.

1.2 Challenges in Deep Learning Clusters

As discussed above, existing DL clusters ignore the features of DL jobs and reuse the resource management techniques from traditional big-data clusters. The mismatches between DL jobs and resource management techniques in DL clusters lead to the following challenges:

Memory over-allocation for data processing applications causes cluster-wise memory underutilization. In the data preparation step (§1.1.1), the data processing jobs are mostly memory-intensive applications that have to keep their working data in memory for good performance. They will experience rapid performance deteriorations when their working sets do not fully fit in memory (§2.2.2). Therefore, existing clusters often over-allocate memory to those jobs for satisfying their peak usages [51], which results in severe memory underutilization across the cluster.

DL training jobs suffer from long JCTs due to their unpredictable execution time. To minimize the average JCT, job scheduling algorithms (e.g., Shortest Job First (SJF) and Shortest Remaining Time First (SRTF)) often require the information of jobs' (remaining) execution time. For DL training jobs, this information is mostly unknown before job completion. Also, it is infeasible to predict a DL training job's remaining execution time in a production environment (§3.2.2). Therefore, existing DL clusters perform poorly in minimizing the average JCT of DL training jobs. They only provide basic orchestrations, i.e., non-preemptive scheduling of jobs as they arrive. Consequently, jobs suffer from long queuing delays when the cluster is over-subscribed.

Bursty model aggregations in DDL training leave the CPU resource being under-utilized. In existing DL clusters, PS instances of a DDL job are run in containers or virtual machines with a fixed group of CPUs. However, a PS instance usually can not keep its assigned CPUs saturated – model aggregation is inherently bursty (§4.2.1). The model aggregation of a tensor can not start until the completion of its *backward* computation (Figure 1.3b). Therefore, the PS instance must idly wait for workers' local gradients that are generated in *backward* computation.

1.3 Thesis Statement and Contributions

Thesis Statement:

Existing DL clusters have not yet been adapted to the unique characteristics of DL jobs. We characterize DL jobs and design techniques to take advantage of them in DL clusters with improved resource efficiency and application performance.

In this thesis, we propose the following software approaches to meet the challenges imposed by DL jobs. These approaches work in the different software layers of DL clusters (Figure 1.4).

1. Decentralized Memory Disaggregation: To improve cluster-wise memory utilization, we bring the idea of memory disaggregation, where the unused memory in data preparation jobs can be utilized as remote memory by other machines in the DL cluster. With this idea, we design and implement INFINISWAP, a remote memory paging system in the operating system layer (Figure 1.4). INFINISWAP can opportunistically harvest and

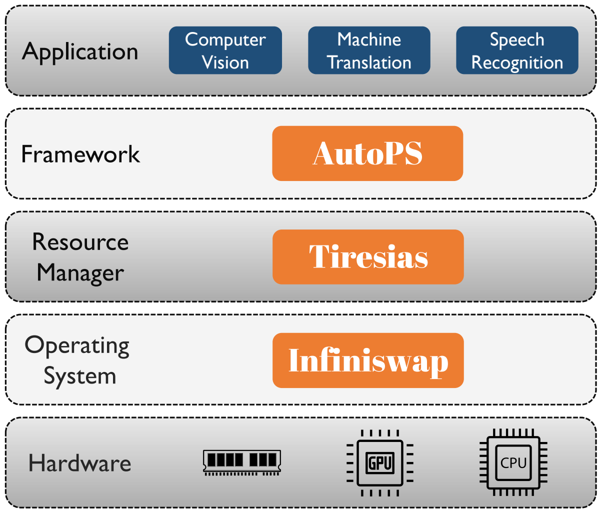


Figure 1.4: INFINISWAP, TIRESIAS, and AUTOPS in a DL cluster.

transparently expose unused memory to remote machines without any modifications in applications, OSes, or hardware. To manage remote memory at scale, INFINISWAP leverages the power-of-many-choices algorithms to perform remote memory mapping and eviction in a decentralized manner. We have deployed and evaluated INFINISWAP on a real cluster with a Remote Direct Memory Access (RDMA) network. INFINISWAP can significantly improve the memory utilization of the cluster and bring performance benefits to the data preparation jobs.

2. Two-Dimensional Attained-Service-Based Scheduler: Although the execution time of a DL training job is unpredictable, its attained service (e.g., execution time) is easy to measure. We propose a two-dimensional attained-service-based scheduler that aims to minimize the overall JCT of DL training jobs without complete job information. Considering the huge variations in both temporal (i.e., executed time) and spatial (i.e., GPU requirements) aspects of DL training jobs, we define the total executed GPU time as the two-dimensional attained service of jobs and use it as the input of our scheduler. This scheduler has two algorithms: *Two-Dimensional Gittins Index* relies on partial information and *Two-Dimensional LAS* is information-agnostic. We implemented TIRESIAS, a GPU cluster manager that can efficiently schedule and place DL training jobs. The profile-based placement algorithm in TIRESIAS relaxes the consolidated placement constraint of DDL training jobs without any user input. Experiments on the Michigan ConFlux cluster and large-scale trace-driven simulations show that TIRESIAS improves the average JCT by up to $5.5\times$ over an Apache YARN-based resource manager used in production. Compared to a state-of-the-art DDL cluster scheduler (Gandiva [164]), TIRESIAS shows significant improvements in the average JCT.

3. Elastic Model Aggregation Service: To reduce the CPU cost of model aggregation, we propose the idea of elastic model aggregation service. It decouples the function of model aggregation from individual training jobs and provides a shared service to execute model aggregations from all the DDL jobs in the cluster. In this service, model aggregations from different jobs are efficiently packed together and dynamically migrated to fit into the available CPUs. Furthermore, it can elastically scale up or down its CPUs based on its total load. We implemented this idea into a new model aggregation framework AUTOPS and evaluated it in testbed experiments and trace-driven simulations. AUTOPS reduces up to 75% of CPU consumption with little or no performance impact on the training jobs. The design of AUTOPS is transparent to the users and can be incorporated into popular DL frameworks.

1.4 Thesis Structure

The remainder of this thesis is organized as follows. Chapter II introduces how to harvest the unused memory resource in a cluster for accelerating data processing applications in DL through INFINISWAP. Chapter III presents TIRESIAS, a GPU cluster manager that aims at reducing the average JCT of DL training job without complete job information. Chapter IV discusses AUTOPS, a unified model aggregation service for DDL training with the purpose of reducing CPU cost. In Chapter V, we conclude this thesis and discuss future work.

CHAPTER II

INFINISWAP: Decentralized Memory Disaggregation for Data Processing

2.1 Introduction

In data preparation phase (§1.1.1), the data processing jobs are mostly memory-intensive applications who keep their working data in memory for good performance. However, they will experience rapid performance deteriorations when their working sets do not fully fit in memory (§2.2.2).

There are two primary ways of mitigating this issue: (i) rightsizing memory allocation and (ii) increasing the effective memory capacity of each machine. Rightsizing is difficult because applications often overestimate their requirements [145] or attempt to allocate for peak usage [51], resulting in severe underutilization and unbalanced memory usage across the cluster. Our analysis of two large production clusters shows that more than 70% of the time there exists severe imbalance in memory utilizations across their machines (§2.2.3).

Proposals for memory disaggregation [91, 143, 77, 110] acknowledge this imbalance and aim to expose a global memory bank to all machines to increase their effective memory capacities. Recent studies suggest that modern RDMA networks can meet the latency requirements of memory disaggregation architectures for numerous in-memory workloads [77, 143]. However, existing proposals for memory disaggregation call for new architectures [91, 19, 15], new hardware designs [116, 117], and new programming models [131, 142], rendering them infeasible.

In this chapter, we present INFINISWAP, a new scalable, decentralized remote memory paging solution that enables efficient memory disaggregation. It is designed specifically for RDMA networks to perform remote memory paging when applications cannot fit their working sets in local memory. It does so *without* requiring any coordination or modifications to the underlying infrastructure, operating systems, and applications (§2.3).

INFINISWAP is not the first to exploit memory imbalance and disk-network latency gap for remote memory paging [74, 76, 121, 132, 56, 47, 71, 115, 26]. However, unlike existing solutions, it does not incur high remote CPU overheads, scalability concerns from central coordination to find machines with free memory, and large performance loss due to evictions from, and failures of, remote memory.

INFINISWAP addresses these challenges via two primary components: a block device that is used as the swap space and a daemon that manages remotely accessible memory. Both are present in every machine and work together without any central coordination. The INFINISWAP block device writes synchronously to remote memory for low latency and asynchronously to disk for fault-tolerance (§2.4). To mitigate high recovery overheads of disks in the presence of remote evictions and failures, we divide its address space into fixed-size slabs and place them across many machines' remote memory. As a result, remote evictions and failures only affect the performance of a fraction of its entire address space.

To avoid coordination, we leverage power of two choices [129] to find remote machines with available free memory. All remote I/O happens via RDMA operations.

The INFINISWAP daemon in each machine monitors and preallocates slabs to avoid memory allocation overheads when a new slab is mapped (§2.5). It also monitors and proactively evicts slabs to minimize performance impact on local applications. Because swap activities on the hosted slabs are transparent to the daemon, we leverage power of many choices [138] to perform batch eviction without any central coordination.

We have implemented INFINISWAP [1] on Linux kernel 3.13.0 (§2.6) and deployed it on a 56 Gbps, 32-machine RDMA cluster on CloudLab [7]. We evaluated it using multiple unmodified memory-intensive applications: VoltDB, Memcached, PowerGraph, GraphX, and Apache Spark using industrial benchmarks and production workloads (§2.7). Using INFINISWAP, throughputs improve between $4\times$ $(0.94\times)$ and $15.4\times$ $(7.8\times)$ over disk (Mellanox nbdX [26]), and median and tail latencies by up to $5.4\times$ $(2\times)$ and $61\times$ $(2.3\times)$, respectively. Memory-heavy workloads experience limited performance difference during paging, while CPU-heavy workloads experience some degradation. In comparison to nbdX, INFINISWAP does not use any remote CPU and provides a $2\times$–$4\times$ higher read/write bandwidth. INFINISWAP can recover from remote evictions and failures while still providing higher application-level performance in comparison to disks. Finally, its benefits hold in the presence of high concurrency and at scale, with a negligible increase in network bandwidth usage.

Despite its effectiveness, INFINISWAP cannot transparently emulate memory disaggregation for CPU-heavy workloads such as Spark and VoltDB (unlike memory-intensive

[1] https://github.com/SymbioticLab/infiniswap

Memcached and PowerGraph) due to the inherent overheads of paging (e.g., context switching). We still consider it useful because of its other tangible benefits. For example, when working sets do not fit in memory, VoltDB's performance degrades linearly using INFINISWAP instead of experiencing a super-linear drop.

We discuss related work in Section 2.9.

2.2 Motivation

This section overviews necessary background (§2.2.1) and discusses potential benefits from paging to remote memory in memory-intensive workloads (§2.2.2) as well as opportunities for doing so in production clusters (§2.2.3).

2.2.1 Background

Paging. Modern operating systems (OSes) support virtual memory to provide applications with larger address spaces than physically possible, using fixed-size pages (typically 4KB) as the unit of memory management. Usually, there are many more virtual pages than physical ones. Page faults occur whenever an application addresses a virtual address, whose corresponding page does not reside in physical memory. Subsequently, the virtual memory manager (VMM) consults with the page table to bring that page into memory; this is known as *paging in*. To make space for the new page, the VMM may need to *page out* one or more already existing pages to a block device, which is known as the *swap space*.

Any block device that implements an expected interface can be used as a swap space. INFINISWAP is written as a virtual block device to perform this role.

See [22] for a detailed description of memory management and its many optimizations in a modern OS.

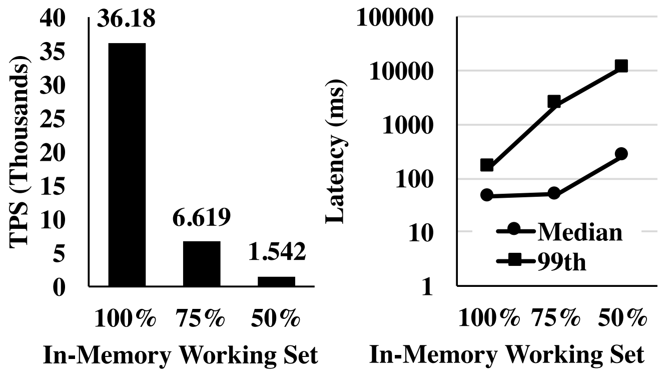
Application Deployment Model. We consider a container-based application deployment model, which is common in production datacenters [161, 145, 51] as well as in container-as-a-service (CaaS) models [94, 159, 21, 11]. These clusters use resource allocation or scheduling algorithms [51, 150, 85, 78] to determine resource shares of different applications and deploy application processes in containers to ensure resource isolation. Applications start paging when they require more memory than the memory limits of their containers.

Network Model. INFINISWAP requires a low-latency, RDMA network, but we do not make any assumptions about specific RDMA technologies (e.g., Infiniband vs. RoCE) or network diameters. Although we evaluate INFINISWAP in a small-scale environment, recent results suggest that deploying RDMA (thus INFINISWAP) on large datacenters may indeed be feasible [175, 90, 128].

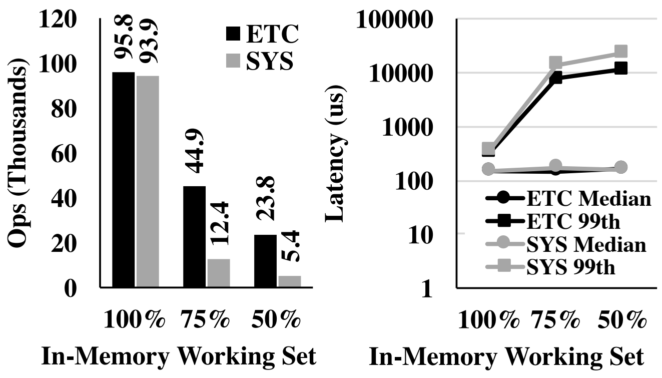
2.2.2 Potential Benefits

To illustrate the adverse effects of paging, we consider four application types: (i) a standard TPC-C benchmark [37] running on the VoltDB [39] in-memory database; (ii) two Facebook-like workloads [49] running on the Memcached [23] key-value store; (iii) PowerGraph [82] running the TunkRank algorithm [38] on a Twitter dataset [106]; and (iv) PageRank running on Apache Spark [168] and GraphX [83] on the Twitter dataset. We found that

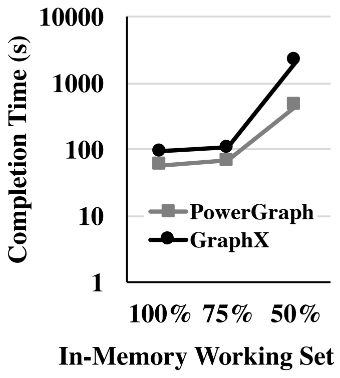
Spark starts thrashing during paging and does not complete in many cases. We defer discussion of Spark to Section 2.7.3.4 and consider the rest here.



(a) TPC-C on VoltDB



(b) Facebook workloads [49] on Memcached



(c) Analytics

Figure 2.1: Performance of in-memory applications under different memory constraints. For modern in-memory applications, a decrease in the percentage of the working set that fits in memory often results in a disproportionately larger loss of performance. This effect is further amplified for tail latencies. All plots show single-machine performance and the median value of five runs. Lower is better for the latency-related plots (lines) and the opposite holds for the throughout-related ones (bars). Note the logarithmic Y-axes in the latency/completion time plots.

To avoid externalities, we only focus on single-server performance. Peak memory usage of each of these runs were around 10GB, significantly smaller than the server's total physical memory. We run each application inside containers of different memory capacities: $x\%$ in the X-axes of Figure 2.1 refers to a run inside a container that can hold at most $x\%$ of the application's working set in memory, and $x < 100$ forces paging. Section 2.7.3

has more details on the experimental setups.

We highlight two observations that show large potential benefits from INFINISWAP. First, paging has significant, non-linear impact on performance (Figure 2.1). For example, a 25% reduction in in-memory working set results in a $5.5\times$ and $2.1\times$ throughput loss for VoltDB and Memcached; in contrast, PowerGraph and GraphX worsen marginally. However, another 25% reduction makes VoltDB, Memcached, PowerGraph, and GraphX up to $24\times$, $17\times$, $8\times$, and $23\times$ worse, respectively.

Second, paging implications are highlighed particularly at the tail latencies. As working sets do not fit into memory, the 99th-percentile latencies of VoltDB and Memcached worsen by up to $71.5\times$ and $21.5\times$, respectively. In contrast, their median latencies worsens by up to $5.7\times$ and $1.1\times$, respectively.

These gigantic performance gaps suggest that a theoretical, 100%-efficient memory disaggregation solution can result in huge benefits, assuming that everything such solutions may require is ensured. It also shows that bridging some of these gaps by a practical, deployable solution can be worthwhile.

2.2.3 Characteristics of Memory Imbalance

To understand the presence of memory imbalance in modern clusters and corresponding opportunities, we analyzed traces from two production clusters: (i) a 3000-machine data analytics cluster (Facebook) and (ii) a 12500-machine cluster (Google) running a mix of diverse short- and long-running applications.

We highlight two key observations – the presence of memory imbalance and its tempo-

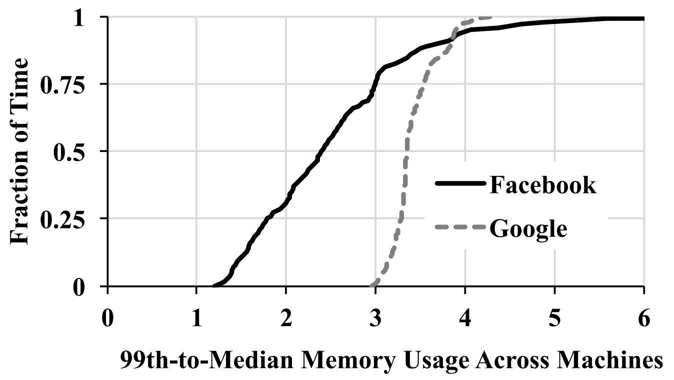


Figure 2.2: Imbalance in 10s-averaged memory usage in two large production clusters at Facebook and Google.

ral variabilities – that guide INFINISWAP's design decisions.

Presence of Imbalance. We found that the memory usage across machines can be substantially unbalanced in the short term (e.g., tens of seconds). Causes of imbalance include placement and scheduling constraints [79, 51] and resource fragmentation during packing [85, 161], among others. We measured memory utilization imbalance by calculating the 99th-percentile to the median usage ratio over 10-second intervals (Figure 2.2). With a perfect balance, these values would be 1. However, we found this ratio to be 2.4 in Facebook and 3.35 in Google more than half the time; meaning, most of the time, more than a half of the cluster aggregate memory remains unutilized.

Temporal Variabilities. Although skewed, memory utilizations remained stable over short intervals, which is useful for predictable decision-making when selecting remote machines. To analyze the stability of memory utilizations, we adopted the methodology described by Chowdhury et al. [60, §4.3]. Specifically, we consider a machine's memory utilization $U_t(m)$ at time t to be stable for the duration T if the difference between $U_t(m)$ and the average value of $U_t(m)$ over the interval $[t, t+T)$ remains within 10% of $U_t(m)$.

We observed that average memory utilizations of a machine remained stable for smaller durations with very high probabilities. For the most unpredictable machine in the Facebook cluster, the probabilities that its current memory utilization from any instant will not change by more than 10% for the next 10, 20, and 40 seconds were 0.74, 0.58, and 0.42, respectively. For Google, the corresponding numbers were 0.97, 0.94, and 0.89, respectively. We believe that the higher probabilities in the Google cluster are due to its long-running services, whereas the Facebook cluster runs data analytics with many short tasks [46].

2.3 INFINISWAP Overview

INFINISWAP is a decentralized memory disaggregation solution for RDMA clusters that opportunistically uses remote memory for paging. In this section, we present a high-level overview of INFINISWAP to help the reader follow how INFINISWAP performs efficient and fault-tolerant memory disaggregation (§2.4), how it enables fast and transparent memory reclamation (§2.5), and its implementation details (§2.6).

2.3.1 Problem Statement

The main goal of INFINISWAP is to *efficiently expose all of a cluster's memory to user applications* without any modifications to those applications or the OSes of individual machines. It must also be *scalable, fault-tolerant, and transparent* so that application performance on remote machines remains unaffected.

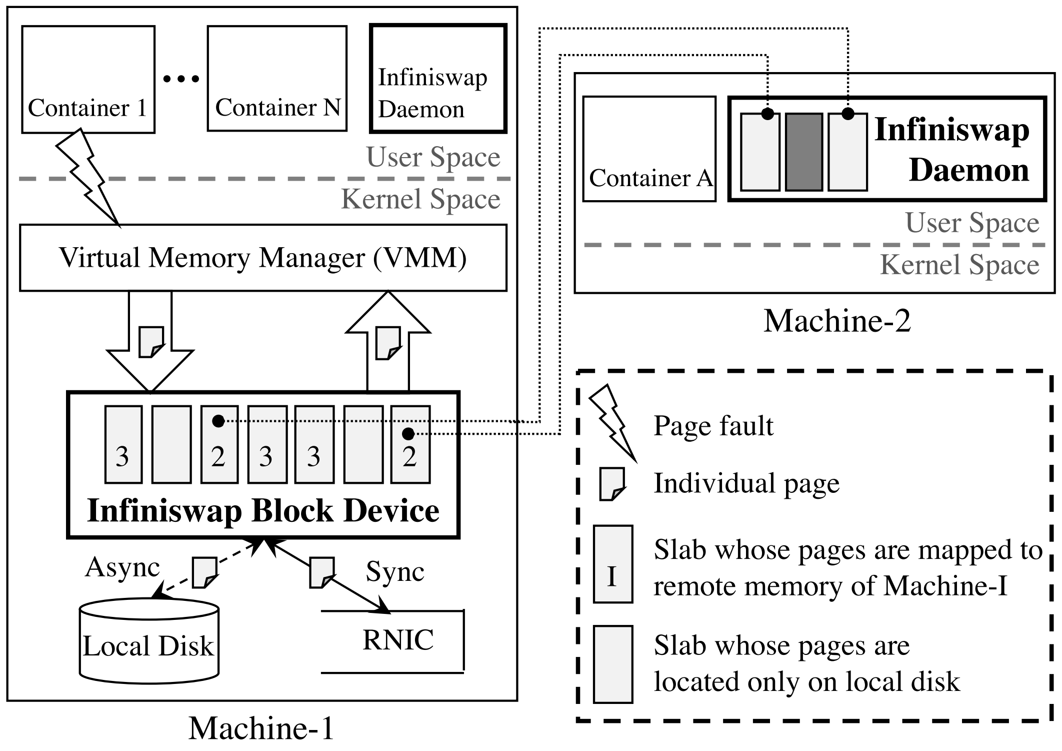


Figure 2.3: INFINISWAP architecture. Each machine loads a block device as a kernel module (set as swap device) and runs an INFINISWAP daemon. The block device divides its address space into slabs and transparently maps them across many machines' remote memory; paging happens at page granularity via RDMA.

2.3.2 Architectural Overview

INFINISWAP consists of two primary components – INFINISWAP block device and INFINISWAP daemon – that are present in every machine and work together without any central coordination (Figure 2.3).

The INFINISWAP block device exposes a conventional block device I/O interface to the virtual memory manager (VMM), which treats it as a fixed-size swap partition. The entire address space of this device is logically partitioned into fixed-size *slabs* (SlabSize). A slab is the unit of remote mapping and load balancing in INFINISWAP. Slabs from the same device can be mapped to multiple remote machines' memory for performance and load balancing (§2.4.2). All pages belonging to the same slab are mapped to the same remote machine. On the INFINISWAP daemon side, a slab is a physical memory chunk of SlabSize that is mapped to and used by an INFINISWAP block device as remote memory.

If a slab is mapped to remote memory, INFINISWAP synchronously writes a page-out request for that slab to remote memory using RDMA WRITE, while writing it asynchronously to the local disk. If it is not mapped, INFINISWAP synchronously writes the page only to the local disk. For page-in requests or reads, INFINISWAP consults the slab mapping to read from the appropriate source; it uses RDMA READ for remote memory.

The INFINISWAP daemon runs in the user space and only participates in control plane activities. Specifically, it responds to slab-mapping requests from INFINISWAP block devices, preallocates its local memory when possible to minimize time overheads in slab-mapping initialization, and proactively evicts slabs, when necessary, to ensure minimal impact on local applications. All control plane communications take place using RDMA SEND/RECV.

We have implemented INFINISWAP as a loadable kernel module for Linux 3.13.0 and deployed it in a 32-machine RDMA cluster. It performs well for a large variety of memory-intensive workloads (§2.7.3).

Scalability. INFINISWAP leverages the well-known power-of-choices techniques [129, 138] during both slab placement in block devices (§2.4.2) and eviction in daemons (§2.5.2). The reliance on decentralized techniques makes INFINISWAP more scalable by avoiding the need for constant coordination, while still achieving low-latency mapping and eviction.

Fault-tolerance. Because INFINISWAP does not have a central coordinator, it does not have a single point of failure. If a remote machine fails or becomes unreachable, INFINISWAP relies on the remaining remote memory and the local backup disk (§2.4.5). If the local disk also fails, INFINISWAP provides the same failure semantic as of today.

2.4 Efficient Memory Disaggregation via INFINISWAP Block Device

In this section, we describe how INFINISWAP block devices manage their address spaces (§2.4.1), perform decentralized slab placement to ensure better performance and load balancing (§2.4.2), handle I/O requests (§2.4.3), and minimize the impacts of slab evictions (§2.4.4) and remote failures (§2.4.5).

2.4.1 Slab Management

An INFINISWAP block device logically divides its entire address space into multiple slabs of fixed size (SlabSize). Using a fixed size throughout the cluster simplifies slab placement and eviction algorithms and their analyses.

Each slab starts in the *unmapped* state. INFINISWAP monitors the page activity rates of each slab using an Exponentially Weighted Moving Average (EWMA) with one second period:

$$A_{\text{current}}(s) = \alpha\, A_{\text{measured}}(s) + (1-\alpha)\, A_{\text{old}}(s)$$

where α is the smoothing factor ($\alpha = 0.2$ by default) and $A(s)$ refers to total page-in and page-out activities for slab s (initialized to zero).

When $A_{\text{current}}(s)$ crosses a threshold (HotSlab), INFINISWAP initiates remote placement (§2.4.2) to map the slab to a remote machine's memory. This late binding helps INFINISWAP avoid unnecessary slab mapping and potential memory inefficiency. We set HotSlab to 20 page I/O requests/second. In our current design, pages are not proactively moved to remote memory. Instead, they are written to remote memory via RDMA WRITE on subsequent page-out operations.

To keep track of whether a page can be found in remote memory, INFINISWAP maintains a bitmap of all pages. All bits are initialized to zero. After a page is written out to remote memory, its corresponding bit is set. Upon failure of a remote machine where a slab is mapped or when a slab is evicted by the remote INFINISWAP daemon, all the bits pertaining to that slab are reset.

In addition to being evicted by the remote machine or due to remote failure, INFINISWAP block devices may preemptively remove a slab from remote memory if $A_{\text{current}}(s)$ goes below a threshold (ColdSlab). Our current implementation does not use this optimization.

2.4.2 Remote Slab Placement

When the paging activity of an unmapped slab crosses the HotSlab threshold, INFINISWAP attempts to map that slab to a remote machine's memory.

The slab placement algorithm has multiple goals. First, it must *distribute slabs from the same block device* across as many remote machines as possible in order to minimize

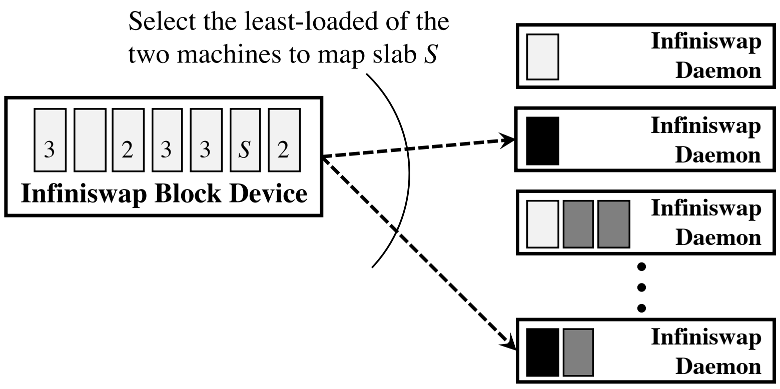


Figure 2.4: Remote machine selection by using power of two choices. INFINISWAP block device uses power of two choices to select machines with the most available memory. It prefers machines without any of its slabs to those who have to distribute slabs across as many machines as possible.

the impacts of future evictions from (failures of) remote machines. Second, it attempts to *balance memory utilization* across all the machines to minimize the probability of future evictions. Finally, it must be *decentralized* to provide low-latency mapping without central coordination.

One can select an INFINISWAP daemon uniformly randomly without central coordination. However, this is known to cause load imbalance [129, 138].

Instead, we leverage power of two choices [129] to minimize memory imbalance across machines. First, INFINISWAP divides all the machines ($\mathbb{M}$) into two sets: those who already have any slab of this block device ($\mathbb{M}_{\text{old}}$) and those who do not ($\mathbb{M}_{\text{new}}$). Next, it contacts two INFINISWAP daemons and selects the one with the lowest memory usage (Figure 2.4). It first selects from $\mathbb{M}_{\text{new}}$ and then, if required, from $\mathbb{M}_{\text{old}}$. The two-step combination distributes slabs across many machines while decreasing load imbalance in a decentralized manner.

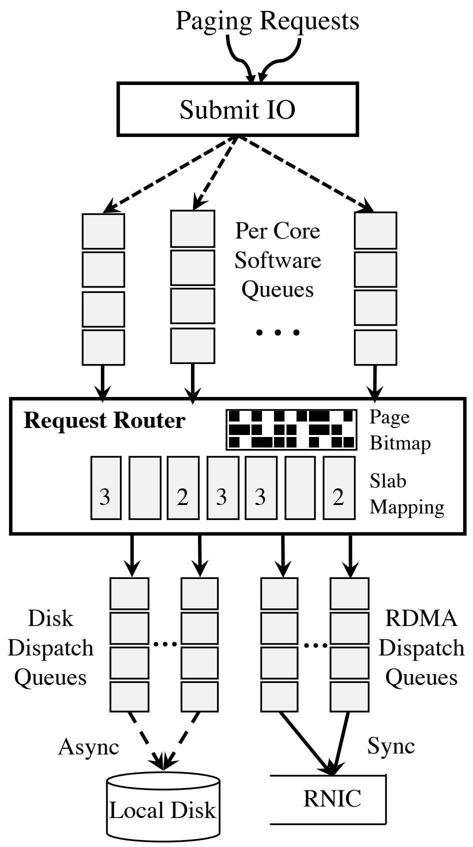


Figure 2.5: INFINISWAP block device overview. Each machine uses one block device as its swap partition.

2.4.3 I/O Pipelines

The VMM submits page write and read requests to INFINISWAP block device using the block I/O interface (Figure 2.5). We use the multi-queue block IO queuing mechanism [5] in INFINISWAP. Each CPU core is configured with an individual software staging queue, where block (page) requests are staged. The request router consults the slab mapping and the page bitmap to determine how to forward them to disk and/or remote memory.

Each RDMA dispatch queue has a limited number of entries, each of which has a registered buffer for RDMA communication. Although the number of RDMA dispatch queues is the same as that of CPU cores, they do not follow one-to-one mapping. Each request from a core is assigned to a random RDMA dispatch queue by hashing its address parameter to avoid load imbalance.

Page Writes. For a page write, if the corresponding slab is mapped, a write request is duplicated and put into both RDMA and disk dispatch queues. The content of the page is copied into the buffer of RDMA dispatch entry, and the buffer is shared between the duplicated requests. Once the RDMA WRITE operation completes, the page write is completed and its corresponding physical memory can be reclaimed by the kernel without waiting for the disk write. However, the RDMA dispatch entry – and its buffer – will not be released until the completion of the disk write operation. When INFINISWAP cannot get a free entry from all RDMA dispatch queues, it blocks until one is released.

For unmapped slabs, a write request is only put into the disk dispatch queue; in this case, INFINISWAP blocks until the completion of the write operation.

Page Reads. For a page read, if the corresponding slab is mapped and the page bitmap is set, an RDMA READ operation is put into the RDMA dispatch queue. When the READ completes, INFINISWAP responds back. Otherwise, INFINISWAP reads it from the disk.

Multi-Page Requests. To optimize I/O requests, the VMM often batches multiple page requests together and sends one multi-page (batched) request. The maximum batch size in the current implementation of INFINISWAP is 128 KB (i.e., 32 4 KB pages). The challenge in handling multi-page requests arises in cases where pages cross slab boundaries, especially when some slabs are mapped and others are unmapped. In these cases, INFINISWAP waits until operations on all the pages in that batch have completed in different sources;

then, it completes the multi-page I/O request.

2.4.4 Handling Slab Evictions

The decision to evict a slab (§2.5.2) is communicated to a block device via the *EVICT* message from the corresponding INFINISWAP daemon. Upon receiving this message, the block device marks the slab as unmapped and resets the corresponding portion of the bitmap. All future requests will go to disk.

Next, it waits for all the in-flight requests in the corresponding RDMA dispatch queue(s) to complete, polling every 10 microseconds. Once everything is settled, INFINISWAP responds back with a *DONE* message.

Note that if $A(s)$ is above the HotSlab threshold, INFINISWAP will start remapping the slab right away. Otherwise, it will wait until $A(s)$ crosses HotSlab again.

2.4.5 Handling Remote Failures

INFINISWAP uses reliable connections for all communication and considers unreachability of remote INFINISWAP daemons (e.g., due to machine failure, daemon process crashes, etc.) as the primary failure scenario. Upon detecting a failure, the workflow is similar to that of eviction: the block device marks the slab(s) on that machine as unmapped and resets the corresponding portion(s) of the bitmap.

The key difference and a possible concern is handling in-flight requests, especially *read-after-write* scenarios. In such a case, the remote machine fails after a page (P) has been written to remote memory but before it is written to disk (i.e., P is still in the disk

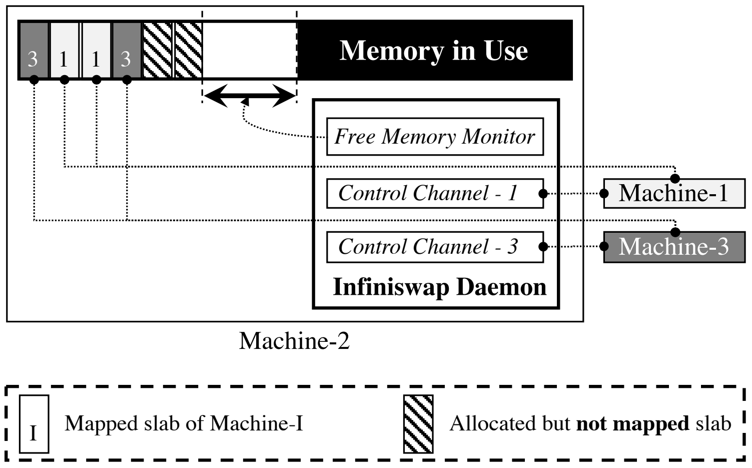


Figure 2.6: INFINISWAP daemon overview. INFINISWAP daemon periodically monitors the available free memory to pre-allocate slabs and to perform fast evictions. Each machine runs one daemon.

dispatch queue). If the VMM attempts to page P in, the bitmap will point to disk, and a disk read request will be added to the disk dispatch queue. Because all I/O requests for the same slab go to the same disk dispatch queue, such read requests will be served by the on-disk data written by the previous write operation.

In the current implementation, INFINISWAP does not handle transient failures separately. A possible optimization would be to use a timeout before marking the corresponding slabs unmapped.

2.5 Transparent Remote Memory Reclamation via INFINISWAP Daemon

In this section, we describe how INFINISWAP daemons (Figure 2.6) monitor and manage memory (§2.5.1) and perform slab evictions to minimize remote and local performance impacts (§2.5.2).

2.5.1 Memory Management

The core functionality of each INFINISWAP daemon is to claim memory on behalf of remote INFINISWAP block devices as well as reclaiming them on behalf of local applications. To achieve this, the daemon monitors the total memory usage of everything else running on the machine using an EWMA with one second period:

$$U_{\text{current}} = \beta\, U_{\text{measured}} + (1 - \beta)\, U_{\text{old}}$$

where β is the smoothing factor ($\beta = 0.2$ by default) and U refers to total memory usage (initialized to 0).

Given the total memory usage, INFINISWAP daemon focuses on maintaining a HeadRoom amount of free memory in the machine by controlling its own total memory allocation at that point. The optimal value of HeadRoom should be dynamically determined based on the amount of memory and the applications running in each machine. Our current implementation does not include this optimization and uses 8 GB HeadRoom by default on 64 GB machines.

When the amount of free memory grows above HeadRoom, INFINISWAP proactively allocates slabs of size SlabSize and marks them as unmapped. Proactive allocation of slabs makes the initialization process faster when an INFINISWAP block device attempts to map to that slab; the slab is marked mapped at that point.

2.5.2 Decentralized Slab Eviction

When free memory shrinks below HeadRoom, INFINISWAP daemon proactively releases slabs in two stages. It starts by releasing unmapped slabs. Then, if necessary, it *evicts* E mapped slabs as described below.

Because applications running on the local machine do not care which slabs are evicted, when INFINISWAP must evict, it focuses on minimizing the performance impact on the machines that are remotely paging. The key challenge arises from the fact that remote INFINISWAP block devices directly interact with their allocated slab(s) via RDMA READ/WRITE operations without any involvement of INFINISWAP daemons. While this avoids CPU involvements, it also prevents INFINISWAP from making any educated guess about performance impact of evicting one or more slabs without first communicating with the corresponding block devices.

This problem can be stated formally as follows. *Given S mapped slabs, how to release the E least-active ones to leave more than* HeadRoom *free memory?*

At one extreme, the solution is simple with global knowledge. INFINISWAP daemon can contact *all* block devices in the cluster to determine the least-used E slabs and evict them. This is prohibitive when E is significantly smaller than the total number of slabs in the cluster. Having a centralized controller would not have helped either, because this would require all INFINISWAP block devices to frequently report their slab activities.

At the other extreme, one can randomly pick one slab at a time without any communication. However, in this case, the likelihood of evicting a busy slab is very high. Consider a parameter $p_b \in [0, 1]$, and assume that a slab is busy (i.e., it is experiencing paging activities

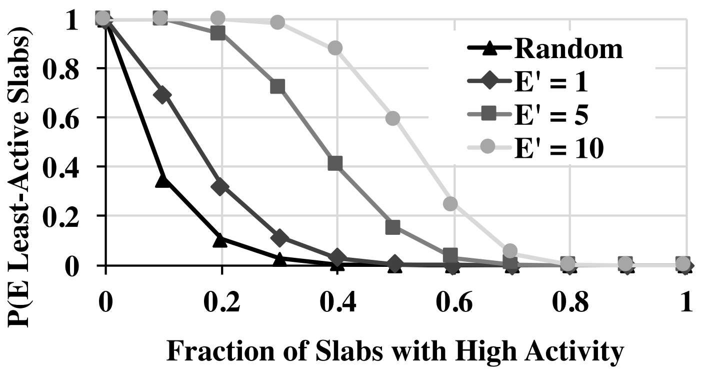


Figure 2.7: Analytical eviction performance for evicting $E(= 10)$ slabs for varying values of E'. Random refers to evicting E slabs one-by-one uniformly randomly.

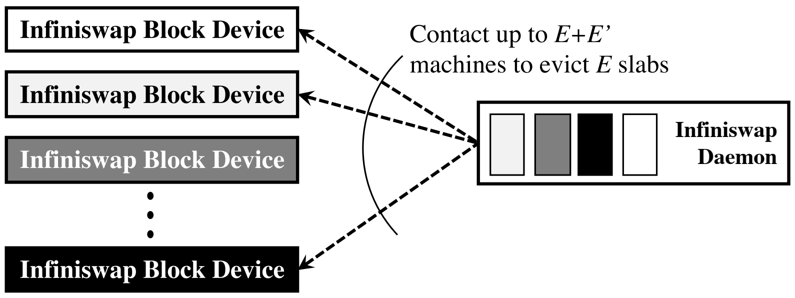


Figure 2.8: Batch eviction by INFINISWAP daemon. INFINISWAP daemon employs batch eviction (i.e., contacting E' more slabs to evict E slabs) for fast eviction of E lightly active slabs.

beyond a fixed threshold) with probability p_b. If we now reformulate the problem to *finding E lightly active slabs* instead of the least-active ones, the probability would be $(1 - p_b)^E$. As the cluster becomes busier (p_b increases), this probability plummets (Figure 2.7).

Batch Eviction. Instead of randomly evicting slabs without any communication, we perform bounded communication to leverage generalized power of choices [138]. Similar techniques had been used before for task scheduling and input selection [137, 160].

For E slabs to evict, INFINISWAP daemon considers $E+E'$ slabs, where $E' \leq E$. Upon communicating with the machines hosting those $E + E'$ slabs, it evicts E least-active ones (i.e., the E slabs of $E + E'$ with the lowest $A(.)$ values) (Figure 2.8). The probability of finding E lightly active slabs in this case is $\sum_{E}^{E+E'} (1 - p_b)^i {p_b}^{E+E'-i} \binom{E+E'}{i}$.

Figure 2.7 plots the effectiveness of batch eviction for different values of E' for $E = 10$. Even for moderate cluster load, the probability of evicting lightly active slabs are significantly higher using batch eviction.

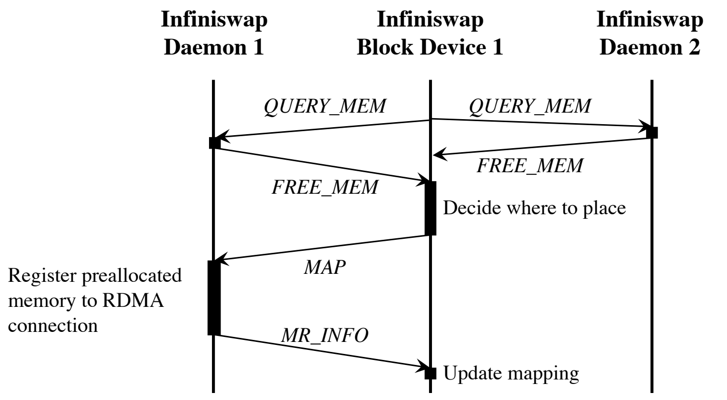
The actual act of eviction is initiated when the daemon sends *EVICT* messages to corresponding block devices. Once a block device completes necessary bookkeeping (§2.4.4), it responds with a *DONE* message. Only then INFINISWAP daemon releases the slab.

2.6 Implementation

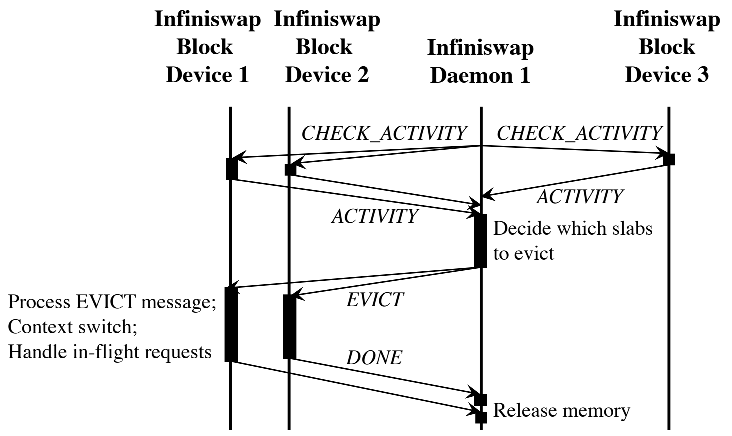
INFINISWAP is a virtual block device that can be used as a swap partition, for example, `/dev/infiniswap0`. We have implemented INFINISWAP as a loadable kernel module for Linux 3.13.0 and beyond in about 3500 lines of C code. Our block device implementation is based on nbdX, a network block device over Accelio framework, developed by Mellanox[26]. We also rely on stackbd [34] to redirect page I/O requests to the disk to handle possible remote failures and evictions. INFINISWAP daemons are implemented and run as user-space programs.

2.6.1 Control Plane

INFINISWAP components use message passing to transfer memory information and memory service agreements. There are eight message types; the first four of them are used by the placement algorithm (Figure 2.9a) and the rest are used by the eviction algorithm (Figure 2.9b).



(a) Sequence diagram during placement

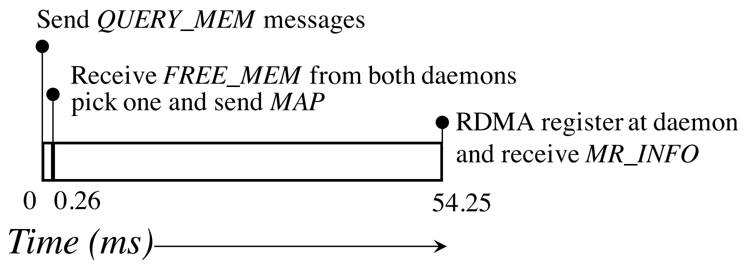

Infiniswap Block Device 1
Infiniswap Block Device 2
Infiniswap Daemon 1
Infiniswap Block Device 3
CHECK_ACTIVITY
CHECK_ACTIVITY
ACTIVITY
ACTIVITY
Decide which slabs to evict
EVICT
Process EVICT message; Context switch; Handle in-flight requests
DONE
Release memory

(b) Sequence diagram during eviction

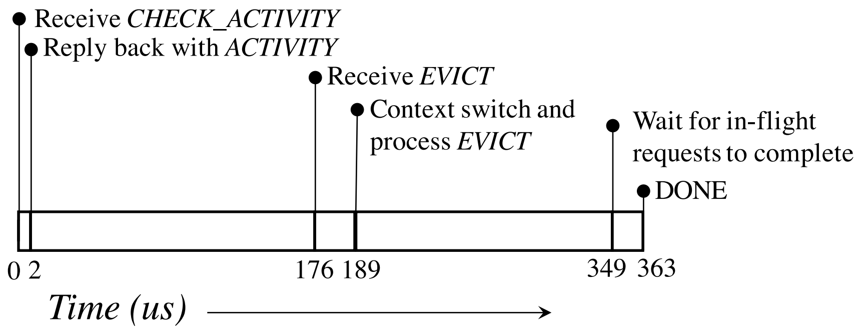
Figure 2.9: Decentralized placement and eviction in INFINISWAP. (a) INFINISWAP block devices use the power of two choices to select machines with the most available memory to place each slab. (b) INFINISWAP daemons use the power of many choices to select slab(s) to evict; in this case, the daemon is contacting 3 block devices to evict 2 slabs.

1. *QUERY_MEM*: Block devices send it to get the number of available memory slabs on the daemon.
2. *FREE_MEM*: Daemons respond to *QUERY_MEM* requests with the number of available memory slabs.
3. *MAP*: Block device confirms that it has decided to use one memory slab from this daemon.

4. *MR_INFO*: Daemon sends memory registration information (`rkey`, `addr`, `len`) of an available memory slab to the block device in response to MAP.
5. *CHECK_ACTIVITY*: Daemons use this message to ask for paging activities of specific memory slab(s).
6. *ACTIVITY*: Block device's response to the *CHECK_ACTIVITY* messages.
7. *EVICT*: Daemons alert the block device which memory slab(s) it has selected to evict.
8. *DONE*: After completely redirecting the requests to the to-be-evicted memory slab(s), block device responds with this message so that the daemon can safely evict and return physical memory to its local OS.


Send *QUERY_MEM* messages
Receive *FREE_MEM* from both daemons pick one and send *MAP*
RDMA register at daemon and receive *MR_INFO*
0
0.26
54.25
Time (ms)

(a) Timing diagram of placement


Receive *CHECK_ACTIVITY*
Reply back with *ACTIVITY*
Receive *EVICT*
Context switch and process *EVICT*
Wait for in-flight requests to complete
DONE
0
2
176
189
349
363
Time (us)

(b) Timing diagram of eviction

Figure 2.10: Timing diagrams (not drawn to scale) during decentralized placement and eviction events.

Detailed information of how they are used during placement and eviction, and corresponding sequence diagrams can be found in Figures 2.9a and 2.9b. Figures 2.10a and 2.10b breakdown placement and eviction, and provide their timings from a INFINISWAP block device's perspective.

2.6.2 Connection Management

INFINISWAP uses reliable connections for all communications. It uses one-sided RDMA READ/WRITE operations for data plane activities; both types of messages are posted by the block device. All control plane messages are transferred using RDMA SEND/RECV operations.

INFINISWAP daemon maintains individual connections for each block device connected to it instead of one for each slab. Similarly, INFINISWAP block devices maintain one connection for each daemon instead of per-slab connections. Overall, for each active block device-daemon pair, there is one RDMA connection shared between the data plane and the control plane.

2.7 Evaluation

We evaluated INFINISWAP on a 32-machine, 56 Gbps Infiniband cluster on CloudLab [7] and highlight the results as follows:

- INFINISWAP provides 2×–4× higher I/O bandwidth than Mellanox nbdX [26]. While nbdX saturates 6 remote virtual cores, INFINISWAP uses none (§2.7.2).

- INFINISWAP improves throughputs of unmodified VoltDB, Memcached, PowerGraph, GraphX, and Apache Spark by up to $4\times$ ($0.94\times$) to $15.4\times$ ($7.8\times$) over disk (nbdX) and tail latencies by up to $61\times$ ($2.3\times$) (§2.7.3).
- INFINISWAP ensures fast recovery from remote failures and evictions with little impact on applications; it does not impact remote applications either (§2.7.4).
- INFINISWAP benefits hold in a distributed setting; it increases cluster memory utilization by $1.47\times$ using a small amount of network bandwidth (§2.7.5).

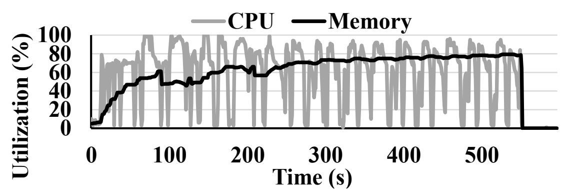
Experimental Setup. Unless otherwise specified, we use SlabSize = 1 GB, HeadRoom = 8 GB, HotSlab = 20 paging activities per second, and $\alpha = \beta = 0.2$ in all the experiments. For comparison, nbdX also utilizes remote memory for storing data.

Each of the 32 machines had 32 virtual cores and 64 GB of physical memory.

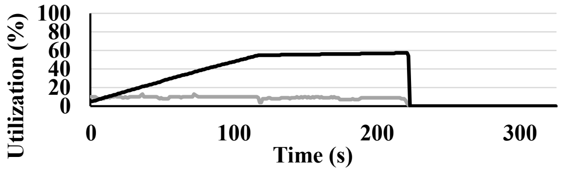
2.7.1 Characteristics of the Benchmarks

We ran each benchmark application in separate containers with 16 GB memory (32 GB only for Spark) and measured their real-time CPU and memory utilizations from cold start. We make the following observations from these experiments.

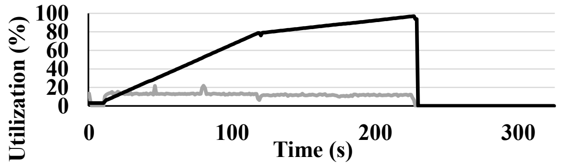
First, while memory utilizations of all applications increased gradually before plateauing, Spark has significantly higher memory utilization along with very high CPU usage (Figure 2.11a). This is perhaps one of the primary reasons why Spark starts thrashing when it cannot keep its working set in memory (i.e., in 75% and 50% configurations). While GraphX exhibits a similar trend (Figure 2.11e), its completion time is much smaller for the same workload.



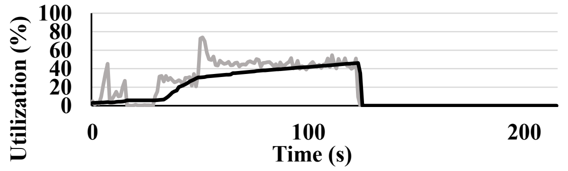
(a) Spark



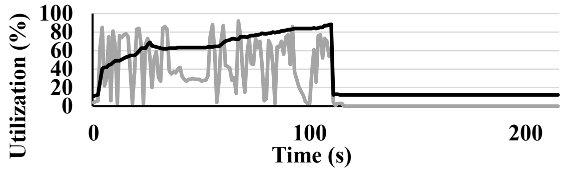
(b) Memcached-ETC



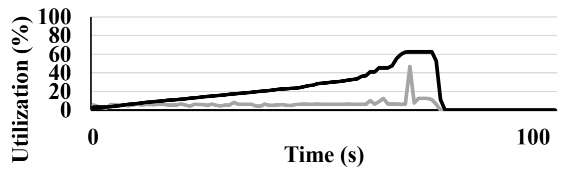
(c) Memcached-SYS



(d) VoltDB



(e) GraphX



(f) PowerGraph

Figure 2.11: CPU and memory usage characteristics of the benchmarked applications. Applications run on containers with 16 GB memory (32 GB only for Spark). Note the increasingly smaller timescales in different X-axes due to smaller completion times of each workload.

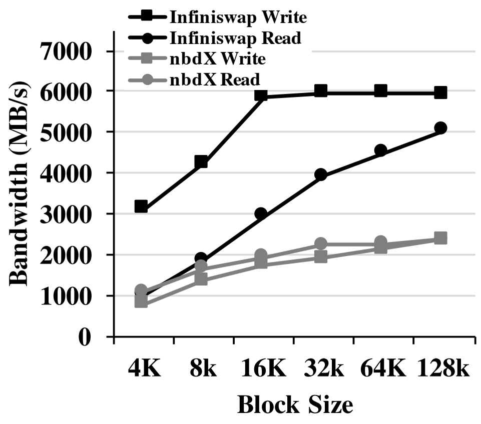
Second, other than Spark and GraphX, VoltDB has at least $3\times$ higher average CPU utilization than other benchmarks (Figure 2.11d). This is one posible explanation of its smaller improvements with INFINISWAP for the 50% and 75% cases in comparison to other less CPU-heavy applications – overheads of paging (e.g., context switch) was possibly a considerable fraction of VoltDB's runtimes.

Third, both ETC and SYS workloads gradually allocate more memory over time, but ETC plateaus early because it has mostly GET operations (Figure 2.11b), whereas SYS keeps increasing because of its large number of SET operations (Figure 2.11c).

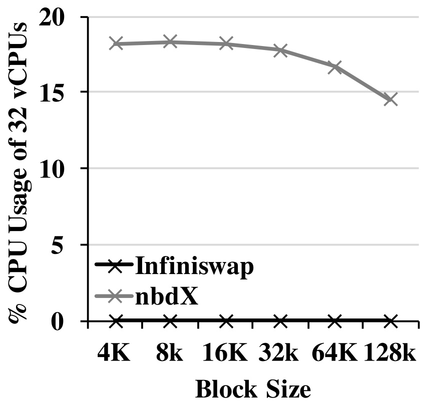
Finally, PowerGraph is the most efficient of the workloads we considered (Figure 2.11f). It completes faster and has the smallest resource usage footprint, both of which contribute to its consistent performances using INFINISWAP across all configurations.

2.7.2 INFINISWAP Performance as a Block Device

Before focusing on INFINISWAP' effectiveness as a decentralized remote paging system, we focus on its raw performance as a block device. We compare it against nbdX and do not include disk because of its significantly lower performance. We used `fio` [9] – a well-known disk benchmarking tool – for these benchmarks.



(a) Bandwidth

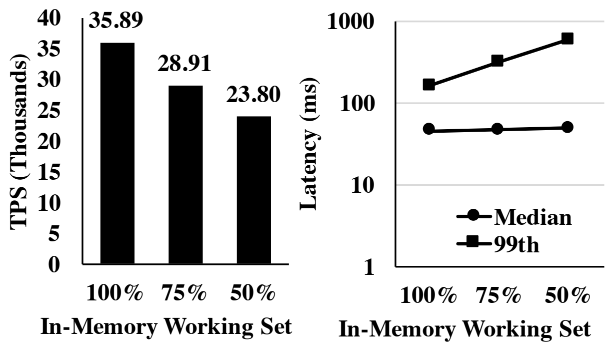


(b) Remote CPU Usage

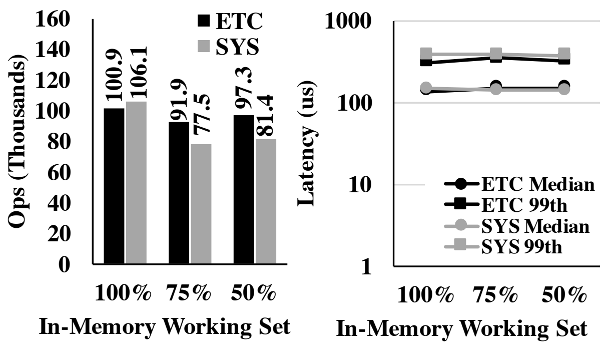
Figure 2.12: Read and write bandwidths of INFINISWAP. INFINISWAP provides higher read and write bandwidths without remote CPU usage, whereas Mellanox nbdX suffers from high CPU overheads and lower bandwidth.

For both INFINISWAP and nbdX, we performed parameter sweeps by varying the number of threads in `fio` from 1 to 32 and I/O depth from 2 to 64. Figure 2.12a shows the highest average bandwidth observed for different block sizes across all these parameter combinations for both block devices. In terms of bandwidth, INFINISWAP performs between $2\times$ and $4\times$ better than nbdX and saturates the 56 Gbps network at larger block sizes.

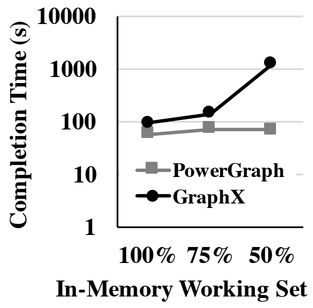
More importantly, we observe that due to nbdX's involvement in copying data to and from RAMdisk at the remote side, it has excessive CPU overheads (Figure 2.12b) and becomes CPU-bound for smaller block sizes. It often saturates the 6 virtual cores it runs on. In contrast, INFINISWAP bypasses remote CPU in the data plane and has close to zero CPU overheads in the remote machine.



(a) TPC-C on VoltDB



(b) Facebook workloads [49] on Memcached



(c) Analytics

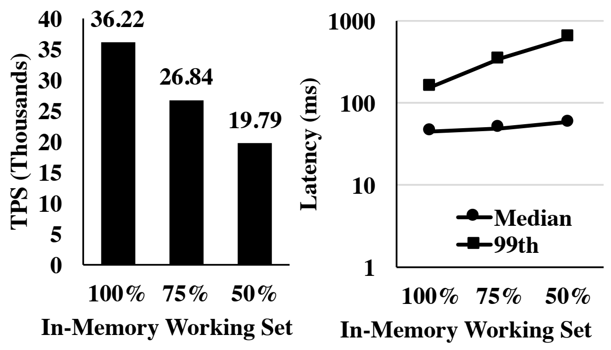
Figure 2.13: Performance of in-memory applications using INFINISWAP. All plots show single-machine, server-side performance, and the median value of five runs. Lower is better for the latency-related plots (lines) and the opposite is true for the throughout-related ones (bars). Note the logarithmic Y-axes in the latency/completion time plots.

2.7.3 INFINISWAP's Impact on Applications

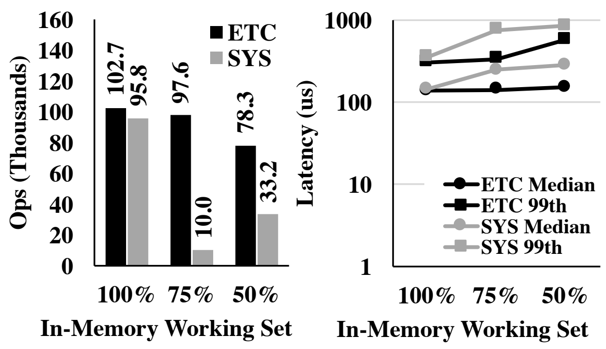
In this section, we focus on INFINISWAP's performance on multiple memory-intensive applications with a variety of workloads (Figure 2.13) and compare it to that of disk (Figure 2.1) and nbdX (Figure 2.14).

Workloads. We used four memory-intensive application and workload combinations:

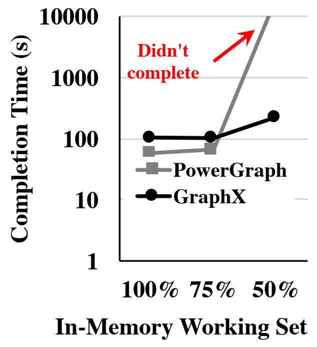
1. TPC-C benchmark [37] on VoltDB [39];
2. Facebook workloads [49] on Memcached [23];
3. Twitter graph [106] on PowerGraph [82]; and
4. Twitter data on GraphX [83] and Apache Spark [168].



(a) TPC-C on VoltDB



(b) Facebook workloads [49] on Memcached



(c) Analytics

Figure 2.14: nbdX performance for comparison. All plots show single-machine, server-side performance, and the median value of five runs. Lower is better for the latency-related plots (lines) and the opposite is true for the throughout-related ones (bars). Note the logarithmic Y-axes in the latency/completion time plots.

Methodology. We focused on single-machine performance and considered three configurations – 100%, 75%, and 50% – for each application. We started with the 100% configuration by creating an `lxc` container with large enough memory to fit the entire workload in memory. We measured the peak memory usage, and then ran 75% and 50% configurations by creating containers with enough memory to fit those fractions of the peak usage. For INFINISWAP and nbdX, we use a single remote machine as the remote swap space.

2.7.3.1 VoltDB

VoltDB is an in-memory, transactional database that can import, operate on, and export large amounts of data at high speed, providing ACID reliability and scalability. We use its community version available on Github.

We use TPC-C to create transactional workloads on VoltDB. TPC-C performs 5 different types of transactions either executed on-line or queued for deferred execution to simulate an order-entry environment. We set 256 warehouses and 8 sites in VoltDB to achieve a reasonable single-container workload of 11.5 GB and run 2 million transactions.

We observe in Figure 2.13a that, unlike disk (Figure 2.1a), performance using INFINISWAP drops linearly instead of super-linearly when smaller amounts of workloads fit in memory. Using INFINISWAP, VoltDB experiences only a $1.5\times$ reduction in throughput instead of $24\times$ using disk in the 50% case. In particular, INFINISWAP improves VoltDB throughput by $15.4\times$ and 99th-percentile latency by $19.7\times$ in comparison to paging to disk. nbdX's performance is similar to that of INFINISWAP (Figure 2.14a).

Overheads of Paging. To understand why INFINISWAP's performance drops significantly when the workload does not fit into memory even though it is never paging to disk, we analyzed and compared its CPU and memory usage with all other considered applications (see § 2.7.1). We believe that because VoltDB is more CPU-intensive than most other memory-intensive workloads we considered, the overheads of paging (e.g., context switches) have a larger impact on its performance.

We note that paging-aware data structure placement (by modifying VoltDB) can help in mitigating this issue [156, 70]. We consider this a possible area of future work.

2.7.3.2 Memcached

Memcached is an in-memory object caching system that provides a simple key-value interface.

We use `memaslap`, a load generation and benchmarking tool for Memcached, to measure performance using recent data published by Facebook [49]. We pick ETC and SYS to explore the performance of INFINISWAP on workloads with different rates of SET operations. Our experiments start with an initial phase, where we use 10 million SET operations to populate a Memcached server. We then perform another set of 10 million queries in the second phase to simulate the behavior of a given workload. ETC has 5% SETs and 95% GETs. The key size is fixed at 16 bytes and 90% of the values are evenly distributed between 16 and 512 bytes [49]. The workload size is measured to be around 9 GB. SYS, on the other hand, is SET-heavy, with 25% SET and 75% GET operations. 40% of the keys have length from 16 to 20 bytes, and the rest range from 20 to 45 bytes. Values of size between 320 and 500 bytes take up 80% of the entire data, 8% of them are smaller, and 12% sit between 500 and 10000 bytes. The workload size is measured to be 14.5 GB. We set the memory limit in Memcached configurations to ensure that for 75% and 50% configurations it will respond using the swap space.

First, we observe in Figure 2.13b that, unlike disk (Figure 2.1b), performance using INFINISWAP remains steady instead of facing linear or super-linear drops when smaller amounts of workloads fit in memory. Using INFINISWAP, Memcached experiences only $1.03\times$ ($1.3\times$) reduction in throughput instead of $4\times$ ($17.4\times$) using disk for the 50% case for the GET-dominated ETC (SET-heavy SYS) workload. In particular, INFINISWAP improves

Memcached throughput by $4.08\times$ ($15.1\times$) and 99th-percentile latency by $36.3\times$ ($61.4\times$) in comparison to paging to disk.

Second, nbdX does not perform as well as it does for VoltDB. Using nbdX, Memcached experiences $1.3\times$ ($3\times$) throughput reduction for the 50% case for the GET-dominated ETC (SET-heavy SYS) workload. INFINISWAP improves Memcached throughput by $1.24\times$ ($2.45\times$) and 99th-percentile latency by $1.8\times$ ($2.29\times$) in comparison to paging to nbdX. nbdX's performance is not very stable either (Figure 2.14b).

Pitfalls of Remote CPU Usage by nbdX. When the application itself is not CPU-intensive, the differences between INFINISWAP and nbdX designs become clearer. As paging activities increase (i.e., for the SYS workload), nbdX becomes CPU-bound in the remote machine; its performance drops and becomes unpredictable.

2.7.3.3 PowerGraph

PowerGraph is a framework for large-scale machine learning and graph computation. It provides parallel computation on large-scale natural graphs, which usually have highly skewed power-law degree distributions.

We run TunkRank [38], an algorithm to measure the influence of a Twitter user based on the number of that user's followers, on PowerGraph. TunkRank's implementation on PowerGraph was obtained from [8]. We use a Twitter dataset of 11 million vertices as the input. The dataset size is 1.3 GB. We use the asynchronous engine of PowerGraph and `tsv` input format with the number of CPU cores set to 2, resulting in a 9 GB workload.

Figure 2.13c shows that, unlike disk (Figure 2.1c), performance using INFINISWAP re-

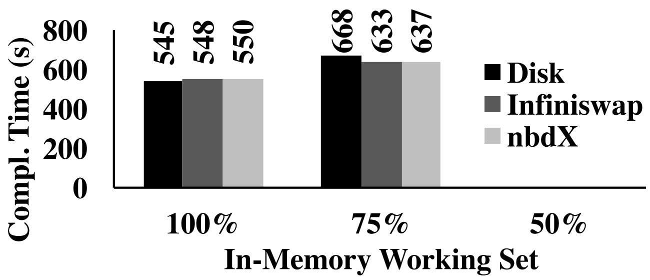


Figure 2.15: Comparative performance for PageRank using Apache Spark. The 50% configuration fails for all alternatives because Spark starts thrashing.

mains stable. Using INFINISWAP, PowerGraph experiences only $1.24\times$ higher completion time instead of $8\times$ using disk in the 50% case. In particular, INFINISWAP improves PowerGraph's completion by $6.5\times$ in comparison to paging to disk.

nbdX did not even complete at 50% (Figure 2.14c).

2.7.3.4 GraphX and Apache Spark

GraphX is a specialized graph processing system built on top of the Apache Spark in-memory analytics engine. We used Apache Spark 2.0.0 to run PageRank on the same Twitter user graph using both GraphX and vanilla Spark. For the same workload, GraphX could run using 12 GB maximum heap, but Spark needed 16 GB.

Figure 2.13c shows that INFINISWAP makes a $2\times$ performance improvement over the case of paging to disk for the 50% configuration for GraphX. However, for Spark, all three of them fail to complete for the 50% configuration (Figure 2.15). In both cases, the underlying engine (i.e., Spark) starts thrashing – applications oscillate between paging out and paging in making little or no progress. In general, GraphX has smaller completion times than Spark for our workload.

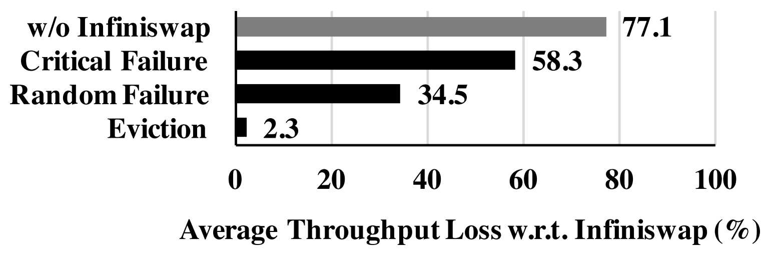


Figure 2.16: Throughput loss of VoltDB. Average throughput loss of VoltDB with 75% in-memory working set w.r.t. INFINISWAP from different failure and eviction events. Lower is better.

2.7.4 Performance of INFINISWAP Components

So far we have considered INFINISWAP's performance without any eviction or failures. In this section, we analyze from both block device and daemon perspectives.

2.7.4.1 INFINISWAP Block Device

For these experiments, we present results for the 75% VoltDB configuration. We select VoltDB because it experienced one of the lowest performance benefits using INFINISWAP. We run it in one machine and distribute its slabs across 6 machines' remote memory. We then introduce different failure and eviction events to measure VoltDB's throughput loss (Figure 2.16).

Handling Remote Failures. First, we randomly turned off one of the 6 machines in 10 different runs; the failed (turned-off) machine did not join back. The average throughput loss was about 34.5% in comparison to INFINISWAP without any failures. However, we observed that the timing of failure has a large impact (e.g., during high paging activity or not). So, to create an adversarial scenario, we turned off one of the 6 machines again, but in this case, we failed the highest-activity machine during its peak activity period. The average throughout loss increased to 58.3%.

Handling Evictions. In this experiment, we evicted 1 slab from one of the remote machines every second and measured the average throughput loss to be about 2.3%, on average, for each eviction event (of 7–29 eviction-and-remapping events). As before, eviction of a high-activity slab had a slightly larger impact than that of one with lower activity.

Time to Map a Slab. We also measured the time INFINISWAP takes to map (for the first time) or remap (due to eviction or failure) a slab. The median time was 54.25 milliseconds, out of which 53.99 went to Infiniband memory registration. Memory registration is essential and incurs the most interfering overhead in Infiniband communication [125]; it includes address translation, and pinning pages in memory to prevent swapping. Note that preallocation of slabs by INFINISWAP daemons mask close to 400 milliseconds, which would otherwise have been added on top of the 54.25 milliseconds. A detailed breakdown of slab mapping is given in Figure 2.9a and 2.10a.

2.7.4.2 INFINISWAP Daemon

Now, we focus on INFINISWAP daemon's reaction time to increase in memory demands of local applications. For this experiment, we set HeadRoom to be 1 GB and started at time zero with INFINISWAP hosting a large number of remote slabs. We started a Memcached server soon after and started performing the ETC workload.

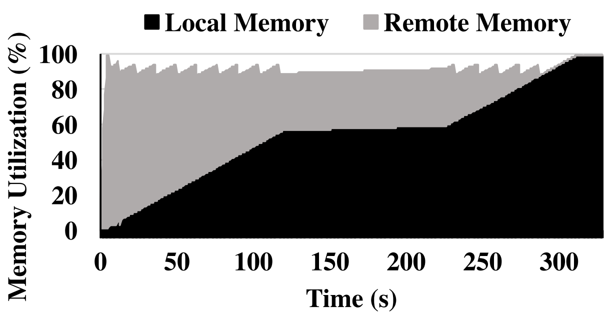


Figure 2.17: Memory utilization when INFINISWAP evicts slabs. INFINISWAP daemon proactively evicts slabs to ensure that a local Memcached server runs smoothly. The white/empty region toward the top represents HeadRoom.

Figure 2.17 shows how INFINISWAP daemon monitored local memory usage and proactively evicted remote slabs to make room – the white/empty region toward the top represents the HeadRoom distance INFINISWAP strived to maintain. After 120 seconds, when Memcached stopped allocating memory for a while, INFINISWAP stopped retreating as well. INFINISWAP resumed evictions when Memcached's allocation started growing again.

Table 2.1: Performance of an in-memory Memcached server with and without INFINISWAP using remote memory.

	ETC		**SYS**	
	W/o	With	W/o	With
Ops (Thousands)	95.9	94.1	96.0	93.5
Median Latency (us)	152.0	152.0	152.0	156.0
99**th Latency (us)**	319.0	318.0	327.0	343.0

To understand whether INFINISWAP retreated fast enough not to have any impact on Memcached performance, we measured its throughput as well as median and 99th-percentile latencies (Table 2.1), observing less than 2% throughput loss and at most 4% increase in tail latency. Results for the SYS workload were similar.

Time to Evict a Slab The median time to evict a slab was 363 microseconds. A detailed breakdown of eviction is provided in Figure 2.9b and 2.10b.

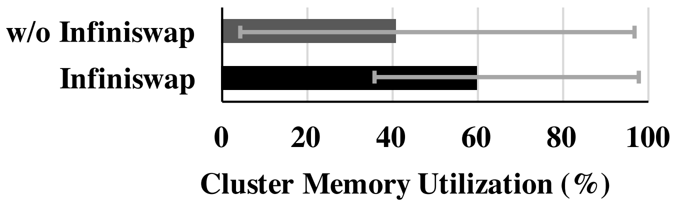
The eviction speed of INFINISWAP daemon can keep up with the rate of memory allocation in most cases. In extreme cases, the impact on application performance can be reduced by adjusting HeadRoom.

2.7.5 Cluster-Wide Performance

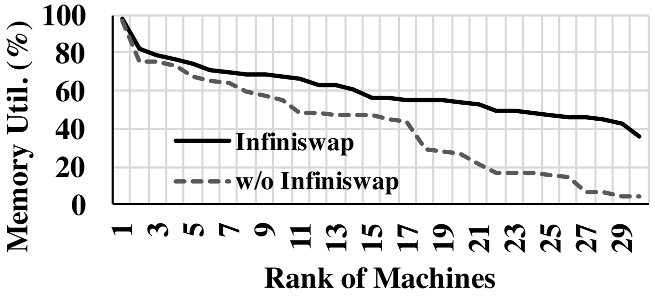
So far we have considered INFINISWAP's performance for individual applications and analyzed its components. In this section, we deploy INFINISWAP on a 32-machine RDMA cluster and observe whether these benefits hold in the presence of concurrency and at scale.

Methodology. For this experiment, we used the same applications, workloads, and configurations from Section 2.7.3 to create about 90 containers. We created an equal number of containers for each application-workload combination. About 50% of them were using the 100% configuration, close to 30% used the 75% configuration, and the rest used the 50% configuration.

We placed these containers randomly across 32 machines to create an memory imbalance scenario similar to those shown in Figure 2.2 and started all the containers at the same time. We measured completion times for the workload running each container; for VoltDB and Memcached completion time translates to transactions- or operations-per-second.



(a) Cluster memory utilization



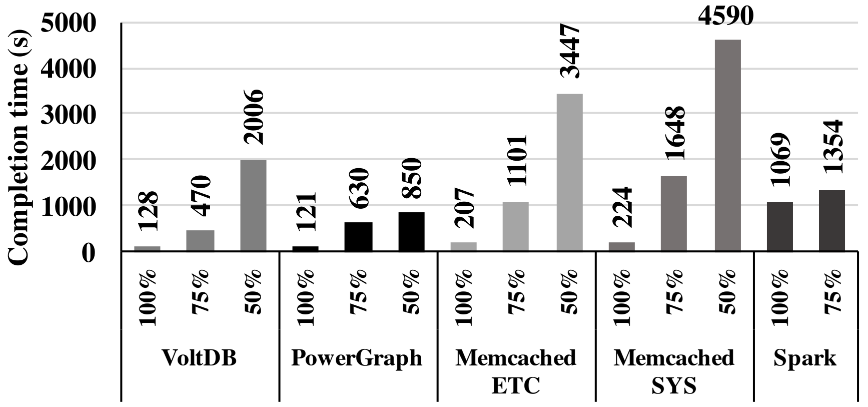
(b) Memory utilization of individual machines

Figure 2.18: Cluster memory utilization when using INFINISWAP. Memory utilization increases and memory imbalance decreases significantly. Error bars in (a) show the maximum and the minimum across machines.

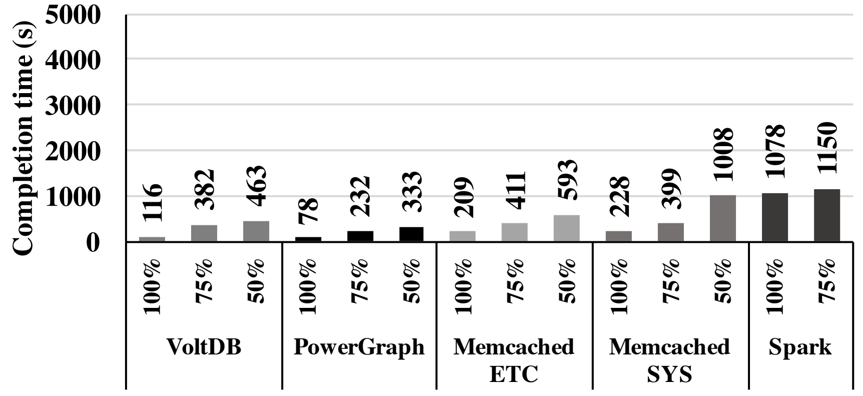
2.7.5.1 Cluster Utilization

Figure 2.18a shows that INFINISWAP increased total cluster memory utilization by $1.47\times$ by increasing it to 60% on average from 40.8%. Moreover, INFINISWAP significantly decreased memory imbalance (Figure 2.18b): the maximum-to-median utilization ratio decreased from $2.36\times$ to $1.6\times$ and the maximum-to-minimum utilization ratio decreased from $22.5\times$ to $2.7\times$.

Increase in Network Utilization. We also measured the total amount of network traffic over RDMA in the case of INFINISWAP. This amounted to less than 1.88 TB over 1300 seconds across 32 machines or 380 Mbps on average for each machine, which is less than 1% of each machine's 56 Gbps interface.



(a) Without INFINISWAP



(b) INFINISWAP

Figure 2.19: Median completion times of containers for different configurations in the cluster experiment. INFINISWAP's benefits translate well to a larger scale in the presence of high application concurrency.

2.7.5.2 Application-Level Performance

Finally, Figure 2.19 shows the overall performance of INFINISWAP. We observe that INFINISWAP's benefits are not restricted only to microbenchmarks, and it works well in the presence of cluster dynamics of many applications. Although improvements are sometimes lower than those observed in controlled microbenchmarks, INFINISWAP still provides $3\times$–$6\times$ improvements for the 50% configurations.

Table 2.2: Cost model parameters.

Parameter	Value
Datacenter OPEX	$0.04/W/month
Electricity Cost	$0.067/kWh
InfiniBand NIC Power	8.41W [17]
InfiniBand Switch Power	231W [18]
Power Usage Effectiveness (PUE)	1.1

2.7.6 Cost-Benefit Analysis

In many production clusters, memory and CPU usages are unevenly distributed across machines (§2.2.3) and resources are often underutilized [173, 146]. Using memory disaggregation via INFINISWAP, machines with high memory demands can use idle memory from other machines, thus enabling more applications to run simultaneously on a cluster and providing more economic benefits. Here we perform a simple cost-benefit analysis to get a better understanding of such benefits.

We limit our analysis only to RDMA-enabled clusters, and therefore, do not consider capital expenditure and depreciation cost of acquiring RDMA hardware. The major source of operational expenditure (OPEX) comes from the energy consumption of Infiniband devices – the parameters of our cost model are listed in Table 2.2. The average cost of INFINISWAP for a single machine is around $\$1.45$ per month.

We also assume that there are more idle CPUs than idle memory in the cluster, and INFINISWAP's benefits are limited by the latter. For example, on average, about 40% and 30% of allocated CPUs and memory are reported to remain unused in Google clusters [146]. We use the price lists from Google Cloud Compute [12], Amazon EC2 [1], and Microsoft Azure [24] to build the benefit model. INFINISWAP is found to increase total cluster

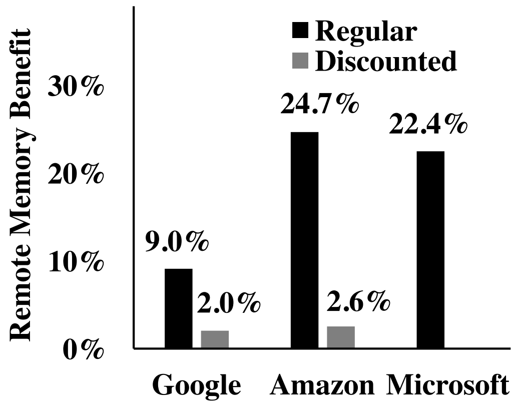


Figure 2.20: Revenue increases with INFINISWAP under three different cloud vendors' regular and discounted pricing models.

memory utilization by around 20% (2.7.5.1), varying slightly across different application deployment scenarios. We assume that there are 20% physical memory on each machine that has been allocated to local applications but the remainder is used as remote memory via INFINISWAP. The additional benefit is then the price of 20% physical memory after deducting the cost of operating Infiniband.

There are several price models from different vendors. In a model we call the *regular pricing model*, resource availability is strictly guaranteed. In another model from Google (preemptible instance) and Amazon (spot instance), resource can be preempted or become unavailable based on resource availability in a cluster. Of course, the resource price in the latter model is much lower than the regular model. We call it the *discounted pricing model*.

If INFINISWAP can ensure unnoticeable performance degradation to applications, remote memory can be counted under regular pricing; otherwise, discounted pricing should be used. Figure 2.20 shows benefits of INFINISWAP. With an ideal INFINISWAP, cluster vendors can gain up to 24.7% additional revenue. If we apply the discounted model, then it decreases to around 2%.

2.8 Discussion

Slab Size. For simplicity and efficiency, unlike some remote paging systems [72], INFINISWAP uses moderately large slabs (SlabSize), instead of individual pages, for remote memory management. This reduces INFINISWAP's meta-data management overhead. However, too large a SlabSize can lower flexibility and decrease space efficiency of remote memory. Selecting the optimal slab size to find a good balance between management overhead and memory efficiency is part of our future work.

Application-Aware Design. Although application transparency in INFINISWAP provides many benefits, it limits INFINISWAP's performance for certain applications. For example, database applications have hot and cold tables, and adapting to their memory access patterns can bring considerable performance benefits [156]. It may even be possible to automatically infer memory access patterns to gain significant performance benefits [70].

OS-Aware Design. Relying on swapping allows INFINISWAP to provide remote memory access without OS modifications. However, swapping introduces unavoidable overheads, such as context switching. Furthermore, the amount of swapped data can vary significantly over time and across workloads even for the same application. Currently, INFINISWAP cannot provide predictable performance without any controllable swap mechanism inside the OS. We would like to explore what can be done if we are allowed to modify OS-level decisions, such as changing its memory allocator or not making context switches due to swapping.

Application Differentiation. Currently, INFINISWAP provides remote memory to all the applications running on the machine. It cannot distinguish between pages from specific applications. Also, there are no limitations in remote memory usage for each application. Being able to differentiate the source of a page will allow us to manage resources better and isolate applications.

Network Bottleneck. INFINISWAP assumes that it does not have to compete with other applications for the RDMA network; i.e., the network is not a bottleneck. However, as the number of applications using RDMA increases, contentions will increase as well. Addressing this problem requires mechanisms to provide isolation in the network among competing applications.

2.9 Related Work

Resource Disaggregation. To decouple resource scaling and to increase datacenter efficiency, resource disaggregation and rack-scale computing have received significant attention in recent years, with memory disaggregation being the primary focus [19, 16, 15, 91, 62, 116, 117]. Recent feasibility studies [143, 77, 110] have shown that memory disaggregation may indeed be feasible even at a large scale, modulo RDMA deployment at datacenter-scale [175, 90]. INFINISWAP realizes this vision in practice and exposes the benefits of memory disaggregation to *any* user application without modifications. Different from INFINISWAP, Fastswap [44], one of the most recent memory disaggregation solutions, modifies the OS kernel for reducing the overhead of memory swap. LegoOS [153] is a new OS designed for the next-generation datacenters who are composed of disaggregated, network-attached hardware (e.g., CPU, memory, storage) components.

Remote Memory. Paging out to remote memory instead of local disks is a known idea [74, 76, 121, 132, 56, 47, 71, 73]. However, their performance and promises were often limited by slow networks and high CPU overheads. Moreover, they rely on central coordination for remote server selection, eviction, and load balancing. INFINISWAP focuses on a decentralized solution for the RDMA environment. HPBD [115] and Mellanox nbdX [26] come the closest to INFINISWAP. Both of them can be considered as network-attached-storage (NAS) systems that use RAMdisk on the server side and are deployed over RDMA networks. However, there are several major differences that make INFINISWAP more efficient, resilient, and load balanced. First, they rely on remote RAMdisks, and data copies to and from RAMdisks become CPU-bound; in contrast, INFINISWAP does not involve remote CPUs, which increases efficiency. Second, they do not perform dynamic memory management, ignoring possibilities of evictions and subsequent issues. Finally, they do not consider fault tolerance nor do they attempt to minimize the impact of failures. Remote Regions [42] provides remove memory access through remote I/O interfaces, which is much simpler than the original *verbs* RDMA interface. However, it requires modifications in applications. Lagar-Cavilla et al. [107] is a software-defined remote memory solution with the objective of cutting down total cost of ownership (TCO) of warehouse-scale computers (WSCs). It proactively move cold data to remote memory and compresses those cold memory pages for reducing memory cost. Leap [122] is a page prefetching solution for remote memory systems. Its prefetching algorithm can improve the page cache hit rate.

Its lightweight data path in kernel for remote paging can greatly reduce the remote access latency.

Software Distributed Shared Memory (DSM). DSM systems [133, 111, 52] expose a shared global address space to user applications. Traditionally, these systems have suffered from communication overheads to maintain coherence. To avoid coherence costs, the HPC community has favored the Partitioned Global Address Space (PGAS) model [65, 53] instead. However, PGAS systems require complete rewriting of user applications with explicit awareness of remote data accesses. With the advent of RDMA, there has been a renewed interest in DSM research, especially via the key-value interface [131, 127, 103, 69, 136, 142]. However, most of these solutions are either limited by their interface or require careful rethinking/rewriting of user applications. INFINISWAP, on the contrary, is a transparent, efficient, and scalable solution that opportunistically leverages remote memory.

2.10 Conclusion

This chapter rethinks the well-known remote memory paging problem in the context of RDMA. We have presented INFINISWAP, a pragmatic solution for memory disaggregation without requiring any modifications to applications, OSes, or hardware. Because CPUs are not involved in INFINISWAP's data plane, we have proposed scalable, decentralized placement and eviction algorithms leveraging the power of many choices. Our in-depth evaluation of INFINISWAP on unmodified VoltDB, Memcached, PowerGraph, GraphX, and

Apache Spark has demonstrated its advantages in substantially improving throughputs (up to $16.3\times$), median latencies (up to $5.5\times$), and tail latencies (up to $58\times$) over disks. It also provides benefits over existing RDMA-based remote memory paging solutions by avoiding remote CPU involvements. INFINISWAP increases the overall memory utilization of a cluster, and its benefits hold at scale.

CHAPTER III

TIRESIAS: A GPU Cluster Manager for Deep Learning

3.1 Introduction

DL is gaining rapid popularity in various domains, such as computer vision, speech recognition, etc. DL training is typically compute-intensive and requires powerful and expensive GPUs. To deal with ever-growing training datasets, it is common to perform DDL training to leverage multiple GPUs in parallel. Many platform providers have built GPU clusters to be shared among many users to satisfy the rising number of DDL jobs [41, 3, 4, 10]. Indeed, our analysis of Microsoft traces shows a $10.5\times$ year-by-year increase in the number of DL jobs since 2016. Efficient job scheduling and smart GPU allocation (i.e., job placement) are the keys to minimizing the cluster-wide average JCT and maximizing resource (i.e., GPU) utilization.

Due to the unique constraints of DDL training, we observe two primary limitations in current cluster manager designs.

1. Naïve scheduling due to unpredictable training time. Although SJF and SRTF algorithms are known to minimize the average JCT [87, 86], they require a job's (remaining) execution time, which is often unknown for DL training jobs. Optimus [140] can predict a DL training job's remaining execution time by relying on its repetitive execution pattern and assuming that its loss curve will converge. However, such proposals make over-simplified assumptions about jobs having smooth loss curves and running to completion; neither is always true in production systems (§3.2.2).

Because of this, state-of-the-art resource managers in production are rather naïve. For example, the internal solution of Microsoft is extended from Apache YARN's Capacity Scheduler that was originally built for big data jobs. It only performs basic orchestration, i.e., non-preemptive scheduling of jobs as they arrive. Consequently, users often experience long queuing delays when the cluster is over-subscribed – up to several hours even for small jobs (§ 3.2.1).

2. Over-aggressive consolidation during placement. Existing cluster managers also attempt to consolidate a DDL job onto the minimum number of servers that have enough GPUs. For example, a job with 16 GPUs requires at least four servers in a 4-GPUs-per-server cluster, and the job may be blocked if it cannot find four completely free servers. The underlying assumption is that the network should be avoided as much as possible because it can become a bottleneck and waste GPU cycles [119]. However, we find that this assumption is only partially valid.

In this chapter, we propose TIRESIAS, a shared GPU cluster manager that aims to address the aforementioned challenges regarding DDL job scheduling and placement (§3.3).

To ensure that TIRESIAS is practical and readily deployable, we rely on the analysis of production job traces, detailed measurements of training various DL models, and two simple yet effective ideas. In addition, we intentionally keep TIRESIAS transparent to users, i.e., all existing jobs can run without any additional user-specified configurations.

Our first idea is a new scheduling framework (2DAS) that aims to minimize the JCT when a DL job's execution time is unpredictable. We propose two scheduling algorithms under this framework: *Discretized 2D-LAS* and *Discretized 2D-Gittins index*. The Gittins index policy [80, 40] is known to be the optimal in the single-server scenario in minimizing the average JCT when JCT distributions are known. Similarly, the classic Least Attained Service (LAS) algorithm [134] has been widely applied in many information-agnostic scenarios, such as network scheduling in datacenters [59, 50]. Both assign each job a priority – the former uses the Gittins index while the latter directly applies the service that job has received so far – that changes over time, and jobs are scheduled in order of their current priorities.

Adapting these approaches to the DDL scheduling problem faces two challenges. First, one must consider both the spatial (how many GPUs) and temporal (for how long) dimensions of a job when calculating its priorities. We show that simply considering one is not enough. Specifically, a job's total attained service in our algorithms jointly considers both its spatial and temporal dimensions.

More importantly, because relative priorities continuously change as some jobs receive service, jobs are continuously preempted. Although this may be tolerable in networking scenarios where starting and stopping a flow is simpler, preempting a DDL job from its GPUs can be expensive because data and model must be copied back and forth between the main memory and GPU memory. To avoid aggressive job preemptions, we apply *priority discretization* atop the two classic algorithms – a job's priority changes after fixed intervals.

Overall, when the cluster manager has the distribution of previous job execution times that may still be valid in the near future, our scheduling framework chooses the *Discretized 2D-Gittins index*. If no prior knowledge is available, *Discretized 2D-LAS* will be applied.

Our second idea is to use model structure to loosen the consolidated placement constraint whenever possible. We observe that only certain types of DL models are sensitive to whether they are consolidated or not, and their sensitivity is due to skew in tensor size distributions in their models. We use this insight to separate jobs into two categories: jobs that are sensitive to consolidation (high skew) and the rest. We implement an RDMA network profiling library in TIRESIAS that can determine the model structure of DDL jobs through network-level activities. By leveraging the profiling library and the iterative nature of DDL training, TIRESIAS can transparently and intelligently place jobs. TIRESIAS first runs the job in a trial environment for a few iterations, and then determines the best placement strategy according to the criteria summarized from previous measurements.

We have implemented TIRESIAS[1] and evaluated using unmodified TensorFlow DDL jobs on a 15-server GPU cluster (each server with four P100 GPUs with NVlink) using traces derived from a Microsoft production cluster. We further evaluate TIRESIAS using large-scale trace-driven simulations. Our results show that TIRESIAS improves the average JCT by up to $5.5\times$ w.r.t. current production solutions and $2\times$ w.r.t. Gandiva [164], a state-of-the-art DDL cluster scheduler. Moreover, it performs comparably to solutions using perfect knowledge of all job characteristics.

[1]https://github.com/SymbioticLab/Tiresias

In summary, we make the following contributions:

- TIRESIAS is the first information-agnostic resource manager for GPU clusters. Also, it is the first that applies two-dimensional extension and priority discretization into DDL job scheduling. It can efficiently schedule and place unmodified DDL jobs without any additional information from the users. When available, TIRESIAS can leverage partial knowledge about jobs as well.
- TIRESIAS leverages a simple, externally-observable, model-specific criteria to determine when to relax worker GPU collocation constraints.
- Our design is practical and readily deployable, with significate performance improvements.

3.2 Background and Motivation

3.2.1 Characteristics of Production DL Cluster

We describe Project Philly [99], the DL cluster manager in one of Microsoft internal production clusters, referred as *P*. *P* is shared by several production teams that work on projects related to a search engine. It is managed by an Apache YARN-like resource manager, which places and schedules DDL jobs submitted by users via a website/REST API front end. *P* supports various framework jobs, including TensorFlow [41], Caffe [6] and CNTK [166]. In 2016, *P* consisted of around 100 4-GPU servers. In 2017, due to the surging demand of running DDL, *P* is expanded by more than 250 8-GPU servers. So, the total number of GPUs has grown by $5\times$. Servers in P are interconnected using a 100-Gbps

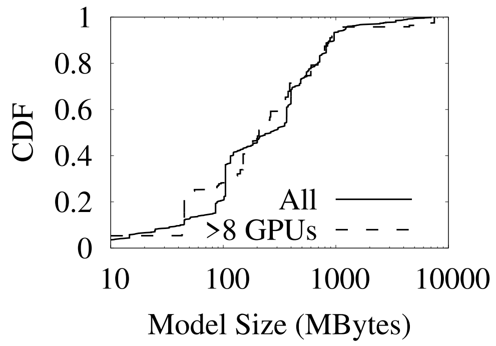
RDMA (InfiniBand) network.

We collect traces from *P* over a 10-week period from Oct. 2017 to Dec. 2017. This cluster runs *Ganglia* monitoring system, which collects per-minute statistics of hardware usage on every server. Since some jobs are quickly terminated because of bugs in user's job configuration, we only show the data of jobs that run for at least one minute. Also, we collect the per-job logs output by the DL framework which include the time for each iteration and the model accuracy along the running time. The network-level activities are monitored by TIRESIAS profiler which is explained in Section 3.3.3.2, that logs every RDMA network operation, e.g., the send and receive of every message,and their timestamps. In addition, we add hooks that intercept the important function calls in a DDL framework, e.g., the start of an iteration or aggregation, and log their timestamps.

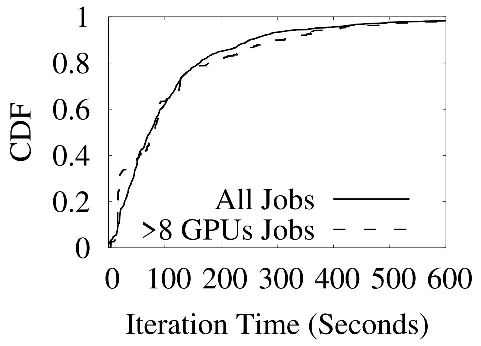
Although we cannot disclose the details of proprietary DL models in *P*, we present the results of several public and popular models, some of which are also run in *P*.

Large and different DL model sizes. As shown in Figure 3.1a, production DL models range from a few hundreds of megabytes to a few gigabytes. The model size distribution is rather independent from the number of GPUs used. According to the cluster users, the number of GPUs used often depends more on the training data volume and the urgency of jobs, and less on model sizes. Larger model sizes mean heavier communication overhead per iteration in distributed training. The largest one is 7.5GB. It may cause network congestion even with 100 Gbps network and greatly hurt the job-level performance. [2] To minimize this overhead, existing job placement strategy intuitively consolidates DDL jobs

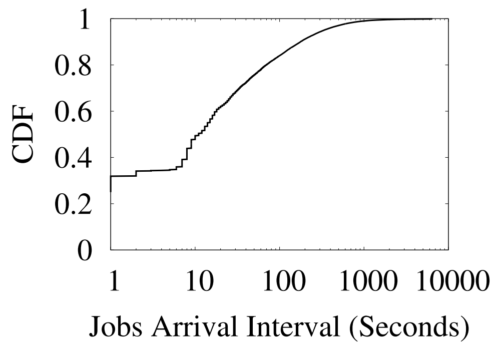
[2]Section 3.3.3 shows that in fact it mostly depends on the model structure.



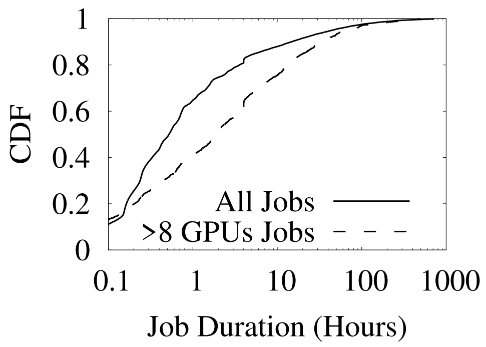
(a) The CDF of DL model sizes being trained.



(b) The CDF of average iteration time per job.



(c) The CDF of DDL job arrival intervals.



(d) The CDF of DDL job duration.

Figure 3.1: Characteristics of DL jobs in the production cluster *P*.

as much as possible.

A staggering increase in the number DDL jobs. We compare the number of DDL jobs (with at least two GPUs) during ten weeks from Oct. 2017 to Dec. 2017, and the number of DDL jobs during the same ten weeks in 2016. The total number of DDL jobs has grown by $10.5\times$ year over year. We refer to jobs using more than 8 GPUs as "large jobs," since such jobs have to run on multiple servers (8 GPUs per server in *P*). Large jobs have grown by $9.4\times$. The largest job run on 128 GPUs in 2017, while the number was 32 GPUs in 2016. We expect this trend to continue as DL jobs are trained on ever larger data sets.

Long job queuing time in production clusters. We also observe that the number of DDL jobs is increasing faster than the speed of cluster expansion. As a result, some jobs have to wait in a queue when the cluster is overloaded. From the trace, we see the average queuing delay of all jobs is 4102 seconds! A brute-force solution is to add GPUs as fast as the demand. However, this poses significant monetary costs – each 8-GPU server in P costs around 100K US Dollars based on public available GPU price. Thus, the DDL cluster service providers are seeking ways to improve job completion time (including the queuing time) given limited GPU resources.

Unpredictable job arrivals. Since the cluster is shared by multiple teams and jobs are submitted on demand, the job arrival intervals are naturally unpredictable. Figure 3.1c shows that the job arrival interval is mostly less than one hour. Many arrival intervals are less than one second, suggesting that they are generated by AutoML to sweeping hyperparameters.

Various aggregation frequency depending on algorithm demands. The communication overhead also depends on how frequently aggregations are performed, which depends on the minibatch sizes. The size of minibatches is determined by the model developers – the larger the minibatches, the larger the learning step, which may help the learning process avoid local optimas but risk final convergence due to too-large steps. Thus, it is usually chosen by the users based on the requirements of specific models.

Figure 3.1b shows that the per iteration time varies significantly across jobs. However, the distribution of large jobs is very close to all jobs. This means that users probably do not

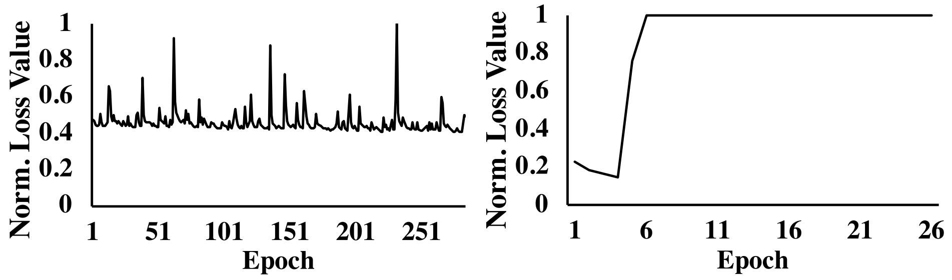


Figure 3.2: Training loss of two production jobs from *P*.

choose minibatch sizes based on how many GPUs are used in each job.

Trial-and-error **exploration.** Training a DL model is not an one-time effort and often works in a *trial-and-error* manner. DL model exposes many *hyperparameters* that express the high-level properties of the model. To get a high-quality model, jobs with different combinations of hyperparameters need to be explored; this is known as *hyperparameter-tuning* [171, 164]. Most of those jobs will be killed because of random errors, or low quality of improvement. We show two representative examples from *P* in Figure 3.2. The spikes in the first example and the non-decreasing curve in the second example lead to early terminations of those jobs. With the feedbacks from trials, users can search new configurations and launch new jobs. Only a very small portion of those jobs with good qualities can run to completion.

3.2.2 Challenges

We highlight three primary challenges faced by DDL cluster managers in production. These challenges originate from the nature of DDL training and are not specific to the Microsoft cluster.

Unpredictable job duration. Current solutions that predict DL job training times [140] all assume DL jobs to (1) have smooth loss curves and (2) reach their training targets and complete. However, for many poor models during a *trial-and-error* exploration, their loss curves are not as smooth as the curves of the best model ultimately picked at the end of exploration, which make it challenging to predict when the training target will be hit. In addition to these proprietary models, popular public models sometimes also show non-smooth curves [93]. Additionally, the termination conditions of DL jobs are non-deterministic. In AutoML [2], most of the trials are killed because of quality issues which are determined by the searching mechanism. Usually, users also specify a maximum epoch number to train for cases when the job cannot achieve the training target. Therefore, a practical resource manager design should not rely on the accuracy/loss curve for predicting eventual job completion time.

Over-aggressive job consolidation. Trying to minimize network communication during model aggregation is a common optimization in distributed training because the network can be a performance bottleneck and waste GPU cycles [119]. Hence, many existing GPU cluster managers blindly follow a consolidation constraint when placing DDL jobs – specifically, they assign all components (parameter servers and workers) of the job to the same or the minimum number of servers. A DDL job will often wait when it cannot be consolidated, even if there are enough spare resources elsewhere in the cluster. Although this constraint was originally set for good performance, it often leads to longer queuing delays and resource under-utilization in practice.

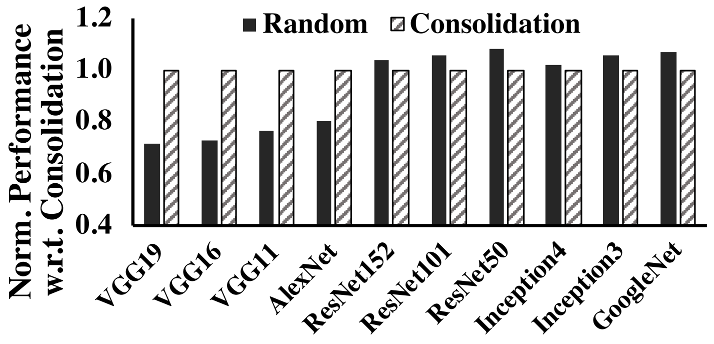


Figure 3.3: Training performance with different placement schemes. Each training job has 8 workers. The performance values are normalized by the value of the consolidation scheme. We use the median value from 10 (20) runs for consolidation (random) scheme.

To understand the importance of this constraint, we run four concurrent 8-GPU jobs using different placement (random and always-consolidate) strategies on eight 4-GPU servers. Similar to [170], each job uses eight parameter servers – the same as the number of workers. Figure 3.3 shows that the locality of workers mainly impacts the VGG family and AlexNet. Nevertheless, neither the cluster operator nor the users can tell which category a job belongs to.

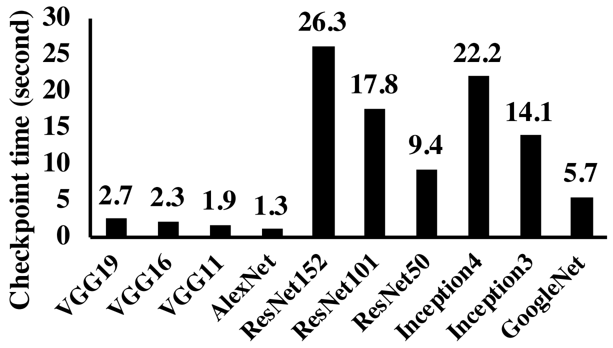


Figure 3.4: Time overhead of pausing a DDL job in Tensorflow. Only the chief worker checkpoints the most updated model.

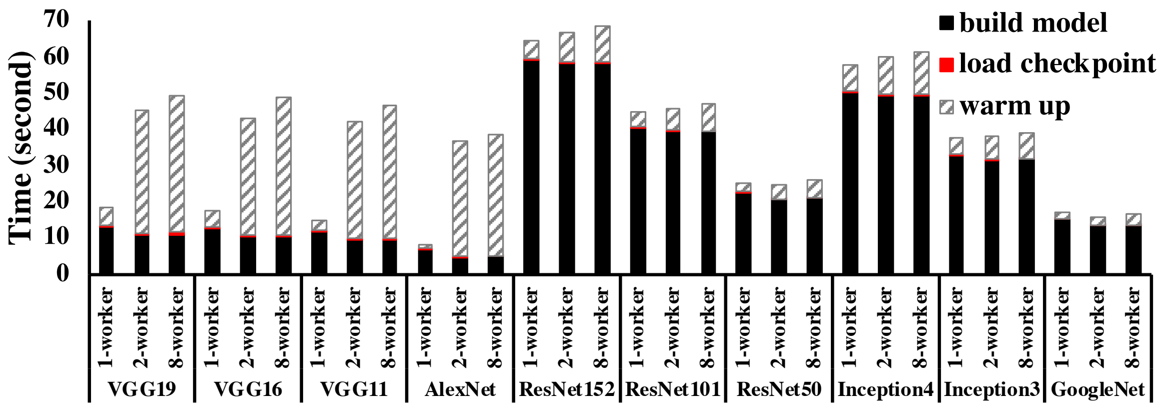


Figure 3.5: Time overhead of resuming a DDL job in Tensorflow. Each model is tested with different number of workers.

Time overhead of preemption. The current production cluster does not preempt jobs because of large time overhead. To show this, we manually test pausing and resuming a DDL job on our local testbed. Upon pausing, the chief worker checkpoints the most recent model on a shared storage. The checkpointed model file will be loaded by all workers when the job is resumed. Figures 3.4 and 3.5 show the detailed numbers. Whenever TIRESIAS preempts a job, we must take this overhead into account.

3.2.3 Potential for Benefits

We can achieve large gains by mitigating two common myths.

Myth I: jobs cannot be scheduled well without exact job duration. Despite the fact that DDL job durations are often unpredictable, their overall distribution can be learned from history logs. The Gittins index policy [80], which is widely used for solving the classic multi-armed bandit problem [80], can decrease the average JCT as long as the job duration distribution is given. Even without that information, the LAS algorithm can efficiently schedule jobs based on their attained service.

Myth II: DDL jobs should always be consolidated. While it is true that consolidated placement of a job may minimize its communication time, we find that some DDL jobs are insensitive to placement. We identify that the core factor is the model structure (§3.3.3).

In the rest of this chapter, we demonstrate that TIRESIAS – using smarter job placement and scheduling strategies – can improve the average total job completion time by more than $5\times$ when running the same set of jobs.

3.3 TIRESIAS Design

This section describes TIRESIAS's architecture, followed by descriptions of its two key components – scheduler and placement manager – and the profiler that learns the job characteristics during runtime.

3.3.1 Overall Architecture

TIRESIAS is a bespoke resource manager for GPU clusters, where the primary workload is DL training. It deals with both allocating GPUs to individual jobs (i.e., job placement) and scheduling multiple jobs over time. So, it has two primary objectives: one user-centric and the other operator-centric.

1. *Minimizing the average JCT:* Jobs should complete as fast as possible regardless of their requirements.
2. *High GPU utilization:* All the GPUs in the cluster should be utilized as much as possi-

ble.

TIRESIAS has an additional goal to balance between operator- and user-centric objectives.

3. *Starvation freedom:* Jobs should not starve for arbitrarily long periods.

Constraints and assumptions: TIRESIAS must achieve the aforementioned objectives under realistic assumptions highlighted in prior sections:

1. *Online job arrival:* Jobs are submitted by users (trial-and-error exploration mechanisms such as AutoML) in an online fashion. The resource requirements of a job J (i.e., the number of parameter servers PS_J and workers W_J) are given but unknown prior to its arrival. Model and data partitions are determined by the DL framework and/or the user [41, 58, 166]. TIRESIAS only deals with resource allocation and scheduling.
2. *Unknown job durations:* Because of non-smooth loss curves and non-deterministic termination in practice, a DL job's duration cannot be predicted. However, the overall distribution of job duration may sometimes be available via history logs.
3. *Unknown job-specific characteristics:* A user does not know and cannot control how the underlying DL framework(s) will assign tensors to parameter servers and the extent of the corresponding skew.
4. *All-or-nothing resource allocation:* Unlike traditional big data jobs where tasks can be scheduled over time [45], DL training jobs require all parameter servers and workers to be simultaneously active; i.e., all required resources must be allocated together.

Job lifecycle: TIRESIAS is designed to optimize the aforementioned objectives without

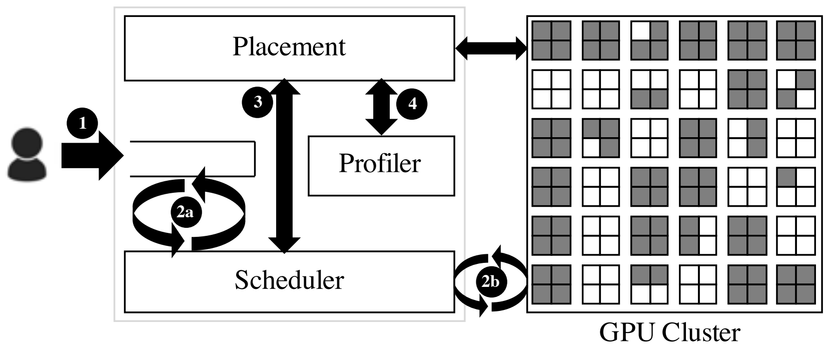


Figure 3.6: TIRESIAS components and their interactions. Job lifecycle under TIRESIAS is described in Section 3.3.1. In this figure, each machine has four GPUs; shaded ones represent GPUs in use.

making any assumptions about a job's resource requirements, duration, or its internal characteristics under a specific DL framework.

Figure 3.6 presents TIRESIAS's architecture along with the sequence of actions that take place during a job's lifecycle. As soon as a job is submitted, its GPU requirements become known, and it is appended to a WAITQUEUE (❶). The scheduler (§3.3.2) periodically schedules jobs from the WAITQUEUE and preempts running jobs from the cluster to the WAITQUEUE (2a and 2b) on events such as job arrival, job completion, and changes in resource availability. When starting a job for the first time or resuming a previously preempted job, the scheduler relies on the placement module (§3.3.3) to allocate its GPUs (❸). If a job is starting for the first time, the placement module first profiles it – the profiler identifies job-specific characteristics such as skew in tensor distribution – to determine whether to consolidate the job or not (❹).

3.3.2 Scheduling

The core of TIRESIAS lies in its scheduling algorithm that must (1) *minimize the average JCT* and (2) *increase cluster utilization* while (3) *avoiding starvation*.

We observe that preemptive scheduling is necessary to satisfy these objectives. One must employ preemption to avoid Head Of Line (HOL) blocking of smaller/shorter jobs by the larger/longer ones – HOL blocking is a known problem of First In, First Out (FIFO) scheduling currently used in production [164]. Examples of preemptive scheduling algorithms include time-sharing,[3] SJF, and SRTF. For example, DL jobs in Gandiva [164] are scheduled by time-sharing. However, time-sharing based algorithms are designed for isolation via fair sharing, not minimizing the average JCT. SJF and SRTF are also inapplicable because of an even bigger uncertainty: it is difficult, if not impossible, to predict how long a DL training job will run. At the same time, size-based heuristics (i.e., how many GPUs a job needs) are not sufficient either, because they ignore job durations.

3.3.2.1 Why Two-Dimensional Scheduling?

By reviewing the time- or sized-based heuristics, we believe that considering only one aspect (spatial or temporal) is not enough when scheduling DDL jobs on a cluster with limited GPU resources. In an SRTF scheduler, large jobs with short remaining time can occupy many GPUs, causing non-negligible queuing delays for many small but newly submitted jobs. If the scheduler is Smallest First (SF) (w.r.t. the number of GPUs), then large jobs may be blocked by a stream of small jobs even if they are close to completion.

To quantify the approaches, we ran trace-driven simulations on three different schedulers using the Microsoft production trace: (1) SF; (2) SRTF; and (3) Shortest Remaining Service First (SRSF). Of them, the first two are single-dimensional schedulers; the last one considers both spatial and temporal aspects. The remaining service in SRSF is the multipli-

[3] Also known as processor-sharing.

Table 3.1: Normalized performance of single-dimensional schedulers w.r.t. SRSF.

	Avg. JCT	Med. JCT	95th JCT
SF	1.52	1.20	3.45
SRTF	1.03	1.01	1.55

cation of a job's remaining time and the number of GPUs. For this simulation, we assume that job durations are given when needed.

Table 3.1 shows that SRSF outperforms the rest in minimizing the average JCT. SRSF has a much smaller tail JCT than the single-dimensional counterparts as well. Altogether, we move forward in building a DDL scheduler that considers both spatial and temporal aspects of resource usage.

Note that, among the three, SF is not a time-based algorithm; hence, it does not actively attempt to minimize the average JCT. As for the rest, SRTF is not always worse than SRSF, either. For example, large-and-short jobs that have many GPUs but short service time can mislead the SRSF scheduler and block many smaller jobs. However, in DL training, multiple GPUs are typically allocated to the jobs that have well-tuned hyperparameters and run to completion. Therefore, the fraction of large-and-short jobs is often small in practice.

3.3.2.2 Two-Dimensional Attained Service-Based Scheduler (2DAS)

We address the aforementioned challenges with the 2DAS scheduler, which schedules DL jobs without relying on their exact durations while taking their GPU requirements into consideration. 2DAS generalizes the classic least-attained service (LAS) scheduling discipline [134] as well as the Gittins index policy [80] to DL job scheduling by considering

Pseudocode 1 Priority function in 2DAS

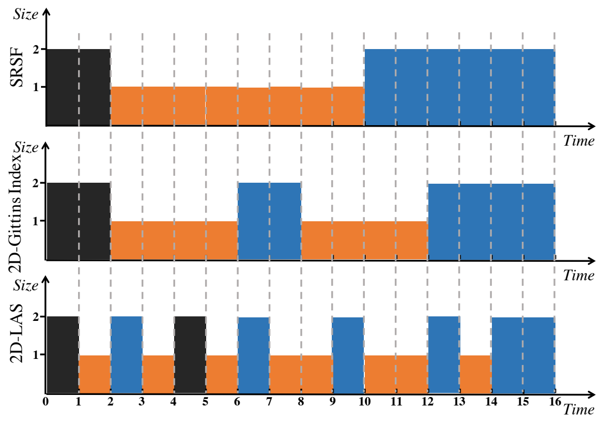
```
procedure PRIORITY(Job $J$, Distribution $\mathbb{D}$)
    if $\mathbb{D}$ is $\varnothing$ then                              ▷ w/o distribution, apply LAS
        $R_J = -W_J \times t_J$
    else                                                 ▷ w/ distribution, apply Gittins index
        $R_J = Gittins\_Index(J, \mathbb{D})$
    end if
    return $R_J$
end procedure

procedure GITTINS_INDEX(Job $J$, Distribution $\mathbb{D}$)
    $a_J = W_J \times t_J$
    $G_J = \sup_{\Delta>0} \frac{\mathbf{P}(S-a_J \leq \Delta | S>a_J)}{\mathbf{E}[min\{S-a_J, \Delta\} | S>a_J]}$
    ▷ P is the probability and E is the mean, both of which are calculated from $\mathbb{D}$. $\Delta$ is the service quantum.
    return $G_J$
end procedure
```

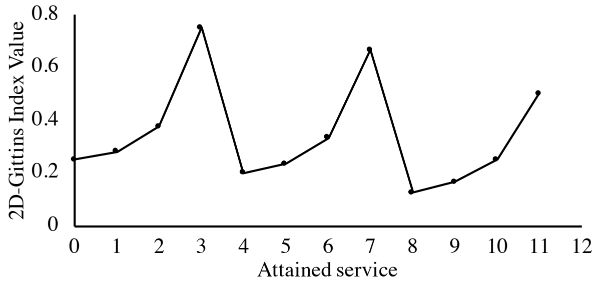
both the *spatial* and *temporal* aspects of such jobs as well as their all-or-nothing characteristic. At a high-level, 2DAS assigns each job a priority based on its attained service. The attained service of a job is calculated based on the number of GPUs it uses (W_J) and the amount of time it has been running so far (t_J). The former becomes known upon the job arrival, while the latter continuously increases.

The priority function in 2DAS can be changed based on different prior knowledge. When *no job duration information* is provided, the priority function applies the LAS algorithm where a job's priority is inverse to its attained service. If the cluster operator provides *the distribution of job duration* from previous experience, then a job's priority equals its Gittins index value (Pseudocode 1). In the Gittins index-based algorithm, the ratio (Line 12) is between (1) the probability that the job will complete within the service quantum of Δ (i.e., the possibility of reward when adding up to Δ overhead on all subsequent jobs) and (2) the expected service that Job J will require for completion.

Both LAS and Gittins index take job's attained service as their inputs. LAS prefers jobs that received less service. All jobs start with the highest priority, and their priorities decrease as they receive more service. The Gittins index value of job represents how *likely* the job that has received some amount of service can complete within the next service quantum. Higher Gittins index value means higher priority.



(a)



(b)

Figure 3.7: A scheduling example using 2D-Gittins index. (a) shows the time sequence of three jobs with three different two-dimensional scheduling algorithms. Job 1 (black) is $(2, 2)$, job 2 (orange) is $(1, 8)$, and job 3 (blue) is $(2, 6)$. The first value in each tuple is the number of GPUs while the second is duration. The scheduling interval is one unit of time. The 2D-Gittins index values for this example are shown in (b). The three jobs have required service 4, 8, and 12, each with probability $1/3$. Job index is used to break ties.

Example: Let us consider an example that illustrates both the algorithms and compares them against SRSF that has complete information (Figure 3.7). Three DL jobs arrive at a two-GPU machine at the same time. The resource requirement of each job is represented using (number of GPUs, duration) pairs. Only SRSF has prior knowledge of job duration. 2D-Gittins index knows the distribution, while 2D-LAS has no related information. The average JCTs in this example are 9.3, 10 and 11.7 units of time for SRSF, 2D-Gittins index, and 2D-LAS, respectively. In general, algorithms with more information perform better in minimizing the average JCT.

3.3.2.3 Priority Discretization

As observed in prior work [59], using continuous priorities can lead to a sequence of preemptions and subsequent resumptions for all jobs. Unlike preempting a network flow or a CPU process, preempting and resuming a DL job on GPU(s) can be time-consuming and expensive (§3.2.2). The excessive cost can make 2DAS infeasible. Furthermore, continuous preemption degenerates 2DAS to fair sharing by time-division multiplexing, which increases the average JCT.

We address these challenges by adopting the *priority discretization* framework based on the classic Multi Level Feedback Queue (MLFQ) algorithm [48, 61, 59].

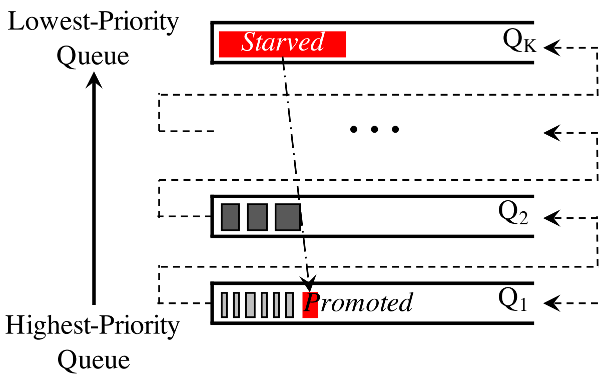


Figure 3.8: Discretized 2DAS with K queues. Starving jobs are periodically promoted to the highest priority queue.

Discretized 2DAS: Instead of using a continuous priority spectrum, we maintain K logical queues ($Q_1, Q_2, \ldots, Q_K$), with queue priorities decreasing from Q_1 to Q_K (Figure 3.8). The i-th queue contains jobs of attained service ($W_J t_J$) values within $[Q_i^{\text{lo}}, Q_i^{\text{hi}})$. Note that $Q_1^{\text{lo}} = 0$, $Q_K^{\text{hi}} = \infty$, and $Q_{i+1}^{\text{lo}} = Q_i^{\text{hi}}$.

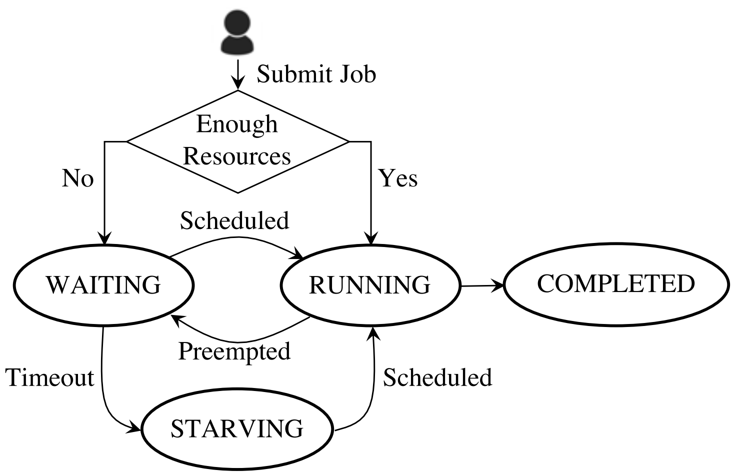


Figure 3.9: State transition diagram of a job in TIRESIAS.

Actions taken during four lifecycle events determine a job's priority (Figure 3.9).

- *Arrival:* If there are available resources, a new job enters the highest priority queue Q_1 when it starts.
- *Activity:* A job is demoted to Q_{i+1} from Q_i, when its ($W_J t_J$) value crosses queue threshold Q_i^{hi}.
- *Starvation:* A job's priority is reset if it had been preempted for too long.
- *Completion:* A job is removed from its current queue upon completion.

The overall structure ensures that jobs with similar ($W_J t_J$) values are kept in the same queue. Jobs with highly different ($W_J t_J$) values are kept in different priority levels.

When LAS is used, jobs in the same queue are scheduled in a FIFO order of their start

Pseudocode 2 2DAS Scheduler

```
1: procedure 2D-LAS(Jobs $\mathbb{J}$, Queues $Q_1 \ldots Q_K$, Distribution $\mathbb{D}$)    ▷ §3.3.2
2:   $\mathbb{P} = \{\}$    ▷ Tracks jobs to preempt
3:   for all Job $J \in \mathbb{J}$ do
4:     if $J$ is RUNNING then
5:       $r_J$ = PRIORITY($J$, $\mathbb{D}$)    ▷ calculate job's priority
6:     end if
7:     if $J$ is WAITING longer than STARVELIMIT then
8:       Reset $t_J$
9:       Enqueue $J$ to $Q_1$    ▷ Promote if $J$ is STARVING
10:    end if
11:  end for
12:  while Cluster has available GPUs do
13:    for all $i \in [1, K]$ do    ▷ Prioritize across queues
14:      if $\mathbb{D}$ is not $\varnothing$ and $i \in [1, K-1]$ then
15:        Sort_Gittins_Index($Q_i$)    ▷ Sort jobs in $Q_i$
16:      end if
17:      for all Job $J \in Q_i$ do    ▷ From the first $J$ in $Q_i$ to the end
18:        if Available GPUs $\geq W_J$ then    ▷ $J$ can run
19:          Mark $W_J$ GPUs as unavailable
20:        else    ▷ $J$ cannot run
21:          $\mathbb{P} = \mathbb{P} \cup J$
22:          Preempt $J$ if it is already RUNNING
23:        end if
24:      end for
25:    end for
26:  end while
27:  for all Job $J \in \mathbb{J}$ and $J \notin \mathbb{P}$ do
28:    if $J$ is not already RUNNING then
29:      if $J$ was not profiled before then
30:        Profile $J$    ▷ §3.3.3.2
31:      end if
32:      Store $J$'s start time    ▷ Used for FIFO in Discretized 2D-LAS
33:      Assign GPUs by comparing $S_J$ to PACKLIMIT    ▷ §3.3.3
34:    end if
35:  end for
36: end procedure
```

time (i.e., when they were first scheduled) without any risk of HOL blocking. Because of the all-or-nothing nature of DDL jobs, high-priority jobs without enough GPUs must be skipped over to increase utilization; as such, FIFO ordering on submission time instead of start time can lead to unnecessary preemptions.

The service quantum Δ in Gittins index is also discretized. For jobs in Q_i, Δ_i equals Q_i^{hi} which is the upper limit of Q_i. When a job consumes all its service quantum, it will be demoted to the lower priority queue. For Gittins index, jobs in the same queue are scheduled according to their Gittins index values. In the last queue, Q_K, Δ_K is set to ∞. In this extreme case, Gittins index performs similar to that of LAS, and jobs in the last queue are scheduled in the FIFO order. The detail of Discretized 2DAS can be found in Pseudocode 2.

Determining K and queue thresholds: While the discretization framework gives us the flexibility to pick K and corresponding thresholds, optimally picking them is an open problem [59, 50]. Instead of frequently solving an integer linear programming (ILP) [50] formulation or using a heavy-weight deep learning mechanism [57], we leverage the classic foreground-background queueing idea [134], which has been shown to perform well for heavy-tailed distributions. There are only two queues ($K = 2$) and only one threshold. Our sensitivity analysis shows that using $K = 2$ performs close to that of larger K values, ignoring preemption overheads. In practice, $K = 2$ limits the number of times a job can be preempted, which reduces job completion time.

Avoiding starvation: Using Discretized 2DAS, some jobs can starve if a continuous stream of small-and-short jobs keep arriving. This is because jobs in the same queue may be skipped over due to the lack of free GPUs. Similarly, jobs in lower priority queues may not receive sufficient GPUs either.

To avoid starvation, we promote a job to the highest-priority Q_1 if it has been WAITING for longer than a threshold: STARVELIMIT (Line 7).

This poses a tradeoff: while promotion can mitigate starvation, promoting too often can nullify the benefits of discretization altogether. To this end, we provide a single knob (PROMOTEKNOB) for the cluster operator to promote a job if its WAITING time so far (δ_J) is PROMOTEKNOB times larger than its execution time so far (t_J); i.e.,

$$\delta_J \geq \text{PROMOTEKNOB} * t_J$$

Setting PROMOTEKNOB $= \infty$ disables promotion and focuses on minimizing the average JCT. As PROMOTEKNOB becomes smaller, 2DAS becomes more fair, sacrificing the average JCT for tail JCT.

Note that both t_J and δ_J are reset to zero to ensure that a promoted job is not demoted right away.

3.3.3 Placement

Given a job J that needs PS_J parameter servers and W_J workers, if there are enough resources in the cluster, TIRESIAS must determine how to allocate them. More specifically, it must determine whether to consolidate the job's GPUs in as few machines as possible or to distribute them. The former is currently enforced in Microsoft production clusters; as a result, a job may be placed in the WAITQUEUE even if there are GPUs available across the cluster.

Taking this viewpoint to its logical extreme, we created an ILP formulation to optimally allocate resources in the cluster to minimize and balance network transfers among machines. The high-level takeaways from such a solution are as follows. First and foremost, it is extremely slow to solve the ILP for a large-scale cluster with many DDL jobs. Second, from small-scale experiments, we found that explicitly minimizing and balancing the load of the network does not necessarily improve DL training performance.

3.3.3.1 ILP Formula for DDL Placement

When placing a DDL job on to a shared GPU cluster, the network traffic generated by that job affects not only itself, but also all the jobs that share the same machines or network links. The existing network status can affect the newly-placed DDL job as well. Therefore, the objective of placing a DDL job is to maximize the overall performance of the entire cluster. To achieve this, we have to minimize the total network traffic and also balance the network load on individual machines in the cluster. In our ILP formulation, the objective function is to minimize the maximal network load of machines when placing a new DDL job onto the cluster.

By default, we assume all DDL jobs have the same number of parameter servers (PS) and GPU worker, which is a common practice [170]. Actually, changing number of parameter servers does not affect the total amount of data in aggregation in the parameter server architecture. There are N GPU nodes in the cluster. N_i is the i-th node whose network traffic from existing DDL jobs is t_i. And N_i has g_i free GPUs before placing any new jobs. We assume a new DDL job J with model size M is going to be placed. There are W GPU workers and K parameter servers in it. The total size of tensors hosted by the j-th parameter server is s_j. For J, w_i is the number of GPU workers placed on N_i. p_{ji} is a binary variable. It will be 1 if the j-th parameter server is placed on N_i, and vice versa.

The total network traffic of N_i comes from three parts: (1) existing traffic, (2) traffic from the workers of J on it, and (3) traffic from the parameter servers of J on it. For collocated parameter servers and workers, the traffic between them has to be deducted. Therefore, the total network traffic T_i is:

$$T_i = t_i + w_i \cdot (M - \sum_{j \in K} p_{ji} \cdot s_j) + \sum_{j \in K} p_{ji} \cdot s_j \cdot (W - w_i)$$

The overall objective can then be expressed as:

$$\text{minimize} \quad \max_{i \in N} \{T_i\}$$

The corresponding constraints are the following:

$$\forall_{i \in N} w_i \leq g_i \tag{III..1}$$

$$\sum_{i \in N} w_i = W \tag{III..2}$$

$$\forall_{j \in K} \sum_{i \in N} p_{ji} = 1 \tag{III..3}$$

The first one is GPU resource constraints on all nodes. The second one requires the consistency of total number of GPU workers in J. The last one means every parameter server must have exactly one host machine. Of course, more constraints, such as CPU and host memory limitations, can be added into this ILP formulation.

How important is consolidation? Given the infeasibility of an ILP-based formulation, we focused on developing a faster solution by asking a simple question: *which jobs benefit* *from consolidation?*

We found that the skew of the model structure (S_J) can be a good predictor. The DL models whose performance are sensitive to consolidated placement (Figure 3.3) have huge tensor(s); their largest tensor size dominates the whole model (Table 3.3). This is because messages sizes in model aggregation are closely related to the structure of the model. For example, a model in TensorFlow consists of many tensors. Each tensor is wrapped as a single communication message.[4] Therefore, the message size distribution in DDL depends on the tensor size distribution of the model. The tensor sizes are often unevenly distributed; sometimes there is a huge tensor which holds most of the parameters in those models. Hence, aggregating larger tensors suffers from network contention more severely, while transmissions of smaller tensors tend to interleave better with each other.

Leveraging this insight, we design TIRESIAS profiler that finds out the skew level of each model, which is then used by the TIRESIAS placement algorithm.

3.3.3.2 Profiler

For a given job J, TIRESIAS's profiler identifies the amount of skew in tensor distributions across parameter servers (S_J) without user input and in a framework-agnostic manner. The skew is a function of the tensor size distribution of the DL job and tensor-to-parameter server mapping of the DL framework (e.g., TensorFlow assigns tensors in a round-robin fashion). Instead of forcing users to design DL models with equal-sized tensors or making assumptions about the tensor assignment algorithm of a given DL framework, we aim to automatically identify the skew via profiling.

[4]Other frameworks may split each tensor into multiple messages, but still, these messages are sent out in clear batches for each tensor.

Because each parameter server periodically sends out its portion of the updated model to each worker (§1.1.2.1), observing these network communications can inform us of the skew. Given that most production DL jobs use RDMA (e.g., InfiniBand in Microsoft) for parameter server-worker communication, and to the best of our knowledge, there exists no RDMA-level traffic monitoring tool, we have built one for TIRESIAS.

TIRESIAS's profiler intercepts communication APIs – including the low-level networking APIs like RDMA `ibverbs` – in each machine to collect process-level communication traces. Whether a DDL job uses RDMA directly or through GPUDirect, TIRESIAS can capture detailed meta-data (e.g., message sizes) about all RDMA communications.

During the profiling run of a job, TIRESIAS aggregates information across all relevant machines to determine S_J for job J. Because each iteration is exactly the same from a communication perspective, we do not have to profile for too many iterations. This predictability also enables us to identify a job's iteration boundaries, model size, and skew characteristics. TIRESIAS's placement algorithm uses this information to determine whether the GPU allocation of a job should be consolidated or not.

3.3.3.3 The Placement Algorithm

TIRESIAS's placement algorithm compares S_J with a threshold (PACKLIMIT); if S_J is larger than PACKLIMIT, TIRESIAS attempts to consolidate the job in as few machines as possible. As explained above, a job with a large skew performs worse due to a skewed communication pattern if it is not consolidated. For the rest, TIRESIAS allocates GPUs in machines to decrease fragmentation. Albeit simple, this algorithm is very effective in practice (§3.5). It performs even better than the previous ILP-based design because the ILP

Table 3.2: Comparison of DL cluster managers.

	YARN-CS	Gandiva [164]	Optimus [140]	TIRESIAS (Gittins index)	TIRESIAS (LAS)
Prior Knowledge	None	None	JCT prediction	JCT distribution	None
Scheduling Algorithm	FIFO	Time sharing	Remaining time driven	Gittins index	LAS
Scheduling Input	Arrival time	N/A	Remaining time	Attained service	Attained service
Schedule Dimensions	Temporal	None	Temporal	Spatial & temporal	Spatial & temporal
Job Priority	Continuous	Continuous	Continuous	Discretized queues	Discretized queues
Job Preemption	N/A	Context switch	Model checkpoint	Model checkpoint	Model checkpoint
Minimizing Average JCT	No	No	Yes	Yes	Yes
Starvation Avoidance	N/A	N/A	Dynamic resource	Promote to Q_1	Promote to Q_1
Job Placement	Consolidation	Trial-and-error	Capacity-based	Profile-based	Profile-based

cannot capture the different effects of consolidation on different models.

Determining PACKLIMIT: We rely on job history to periodically update PACKLIMIT. Currently, we use a simple linear classifier to periodically determine the PACKLIMIT value using a job's placement and corresponding performance as features. More sophisticated mechanism to dynamically determine PACKLIMIT can be an interesting future work.

3.3.4 Summary

Compared to Apache YARN's Capacity Scheduler (YARN-CS) and Gandiva, TIRESIAS aims to minimize the average JCT. Unlike Optimus, TIRESIAS can efficiently schedule jobs without or with partial prior knowledge (Table 3.2). Additionally, TIRESIAS can smartly place DDL jobs based on the model structure automatically captured by the TIRESIAS profiler.

3.4 Implementation

We have implemented TIRESIAS as a centralized resource manager. The Discretized 2DAS scheduler, the placement algorithm, and the profiler are integrated into the central master, and they work together to appropriately schedule and place DDL jobs. Similar to using current DDL clusters, users submit their DDL jobs with the resource requirements, primarily the number of parameter servers (PS_J) and the number of GPUs/workers (W_J). The resource manager then handles everything, from resource allocation when a job starts to resource reclamation when it completes.

As mentioned earlier, TIRESIAS makes job placement decisions based on profiling via a network monitoring library. This library is present in every server of the cluster and communicates with the central profiler so that TIRESIAS can determine the skew of each new DDL job.

Central master: In addition to starting new jobs and completing existing ones, a major function of the master is to preempt running jobs when their (GPU) resources are assigned to other jobs by the scheduler. Because of the iterative nature of DL jobs, we do not need to save all the data in GPU and main memory for job preemption. Currently, we use the checkpoint function provided by almost every DL framework and just save the most updated model for the preempted job. When a preemption is triggered, the job is first paused; then its chief worker checkpoints its model to a cluster-wide shared file system. When a paused job is resumed again by the scheduler, its most recent checkpoint will be loaded before it is restarted. The central master also determines a job's placement using the placement algorithm and the profiler.

Distributed RDMA monitoring: Because RDMA is widely used in GPU clusters for DDL jobs, we implement the profiler as a loadable library that intercepts RDMA `ibverbs` APIs. Therefore, it can record all the RDMA activities on each server, such as building connections, sending and receiving data. The RDMA-level information of all relevant workers and parameter servers are then aggregated at the central profiler. Based on the aggregated information (e.g., message size and the total amount of traffic), TIRESIAS can resolve the detailed model information of a given DDL job, including its skew. Though implemented for RDMA networks, the profiler can easily be extended to support TCP/IP networks by intercepting socket APIs.

3.5 Evaluation

We have deployed TIRESIAS on a 60-GPU cluster and evaluated it using experiments and large-scale simulations using production traces from Microsoft. The highlights are:

- In testbed experiments, TIRESIAS improves the average JCT by up to $5.5\times$ and the makespan by $1.21\times$ compared to YARN-CS. It also performs comparably to SRTF, which uses complete prior information (§3.5.2). TIRESIAS's benefits are due to job placement benefits for skewed DDL jobs and reduction in queueing delays during scheduling.

- TIRESIAS's benefits hold for large-scale simulation of the production trace from Microsoft (§3.5.3).
- TIRESIAS is robust to various configuration parameters and workload variations (§3.5.4).

In this section, TIRESIAS-G (TIRESIAS-L) represents TIRESIAS using the *Discretized 2D-Gittins index* (*Discretized 2D-LAS*).

3.5.1 Experimental Setup

Testbed. Our testbed consists of 15 4-GPU PowerNV 8335-GTB machines from IBM in the Michigan ConFlux cluster. Each machine has 4 NVIDIA Tesla P100 GPUs with 16 GB GPU memory, two 10-core (8 threads per core) POWER8 CPUs, 256 GB DDR4 memory, and a 100 Gbps EDR Mellanox InfiniBand adapter. There is also a high-performance cluster file system, GPFS [149], shared among those machines. In TIRESIAS, the checkpoint files used in job preemptions are written to and read from GPFS. The read and write throughput of GPFS from each machine is 1.2 GB/s.

Simulator. We developed a discrete-time simulator to evaluate TIRESIAS at large scale using a real job trace from Microsoft. It simulates all job events in TIRESIAS, including job arrival, completion, demotion, promotion, and preemption. However, it cannot determine job training time with the dynamic cluster environment; instead, it uses actual job completion times.

Table 3.3: Characteristics of the DL models (in TensorFlow) used the experiments.

Model	Model size (MB)	#Tensors	#Large tensors ($\geq$ 1MB)	Largest tensor size (MB)	Largest tensor ratio
VGG19	548.1	39	15	392.0	71.5%
VGG16	527.8	33	12	392.0	74.3%
VGG11	506.8	23	9	392.0	77.3%
AlexNet	235.9	17	7	144.0	61.0%
ResNet152	230.2	778	48	9.0	3.9%
ResNet101	170.4	523	35	9.0	5.3%
ResNet50	97.7	268	18	9.0	9.2%
Inception4	162.9	599	81	5.9	3.6%
Inception3	91.0	397	21	7.8	8.6%
GoogleNet	26.7	117	7	3.9	14.6%

Workload. Given the scale of our GPU cluster, we generate our experimental workload of 480 DL/DDL jobs by scaling down the original job trace. Job requirements (number of GPUs, and training time) in our workload follow the distributions of the real trace. Half of these jobs are single-GPU DL jobs; the rest are DDL ones (40 2-GPU jobs, 80 4-GPU jobs, 90 8-GPU jobs, 25 16-GPU jobs, and 5 32-GPU jobs). The number of parameter servers in each DDL job is the same as its GPU number. Each model in Table 3.3 has 48 jobs. Each job has a fixed number of iterations to run. The training time of jobs varies from 2 mins to 2 hours. Jobs arrive following a Poisson process with an average inter-arrival time of 30 seconds. We run the jobs in synchronous data parallelism mode using TensorFlow 1.3.1 with RDMA extension and using model files from the TensorFlow benchmark [36].

Job Bins. We category our jobs based on both their spatial (number of GPUs) and temporal (job training time) characteristics (Table 3.4). For the original trace, we consider a job to be *small* if it does not need more than 8 GPUs (Microsoft uses 8-GPU machines) and *short* if its training time is less than 4 hours. After scaling down, we consider a job to be

Table 3.4: Bins of DL jobs with different GPU requirements (**S**mall and **L**arge) and execution time ($\mathbb{S}$hort and $\mathbb{L}$ong).

Bin	1 (**S**$\mathbb{S}$)	2 (**S**$\mathbb{L}$)	3 (**L**$\mathbb{S}$)	4 (**L**$\mathbb{L}$)
% of Jobs	63.5%	12.5%	16.5%	7.5%

small if it needs at most 4 GPUs (we are using 4-GPU machines) and short if it trains for less than 800 seconds.

Baselines. We compare TIRESIAS to an Apache YARN's capacity scheduler (YARN-CS) used in Microsoft (§ 3.2.1). For comparison, we also implement an *SRTF* scheduler that has complete information, i.e., the eventual training time that in practice, cannot be obtained before running the job. Note that job durations are unknown to both TIRESIAS and YARN-CS. SRTF uses TIRESIAS's placement mechanism. We also comapre TIRESIAS with the time-sharing scheduler in Gandiva [164] in the large-scale simulation.

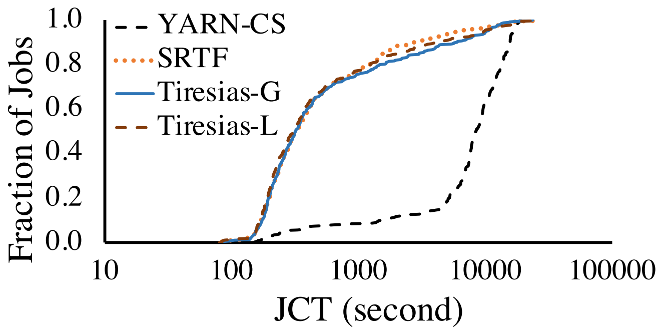
Metric. Our key metric is the improvement in the average JCT (i.e., time from submission to completion):

$$\text{Factor of Improvement} = \frac{\text{Duration of an Approach}}{\text{Duration of T\textsc{iresias}-L}}$$

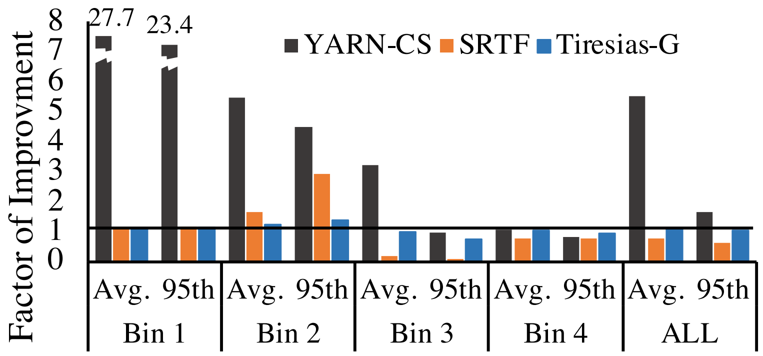
To clearly present the performance of TIRESIAS, unless otherwise specified, the results of all schedulers (including TIRESIAS-G) are normalized by that of TIRESIAS-L. Factor of improvement (FOI) greater than 1 means TIRESIAS-L is performing better, and vice versa.

3.5.2 TIRESIAS in Testbed Experiments

In testbed experiments, we compare the performance of YARN-CS, SRTF, TIRESIAS-G and TIRESIAS-L. For TIRESIAS, there are two priority queues with a threshold of 3200 GPU seconds. The PROMOTEKNOB for avoiding starvation is disabled in Testbed experiments.



(a) Individual job completion times



(b) Summarized results

Figure 3.10: Improvements in the average JCT using TIRESIAS w.r.t. YARN-CS and SRTF.

3.5.2.1 JCT Improvements

TIRESIAS-L achieves $5.5\times$ improvement in terms of the average JCT w.r.t. to YARN-CS (Figure 3.10). If we look at the median JCT, then TIRESIAS-L is $27\times$ better than YARN-CS. TIRESIAS-G has almost the same performance as TIRESIAS-L ($1.06\times$ in average, $1.05\times$ in median). Its negligible performance loss is due to more job preemptions (§3.5.2.4). Half of all jobs avoid severe queueing delays using TIRESIAS. Moreover, TIRESIAS is not far from SRTF either.

The key idea of TIRESIAS's scheduler is avoiding queueing delays to small or short jobs, thus saving them from large or long jobs. When using TIRESIAS-L (TIRESIAS-G), the average JCT of jobs in Bin1 (**S**$\mathbb{S}$) is just 300 (330) seconds, which is $27.6\times$ ($25.2\times$) better than that using YARN-CS. On the other hand, jobs in Bin4 (**L**$\mathbb{L}$) have almost the same average JCT in both TIRESIAS and YARN-CS.

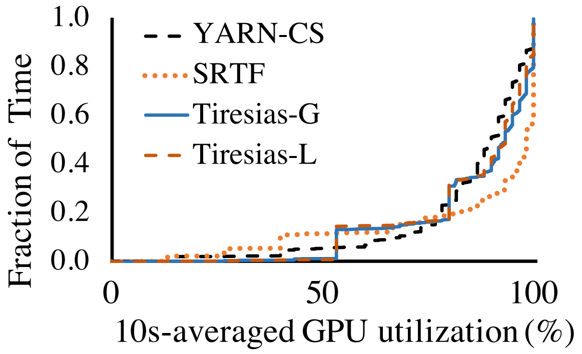


Figure 3.11: Cluster-wide GPU utilization.

3.5.2.2 Cluster-Wide GPU Utilization

Figure 3.11 shows the averaged GPU utilizations of our cluster over time. While there are some small variations, overall utilizations across solutions look similar. However, TIRESIAS reduces the makespan compared to YARN-CS. The makespan of TIRESIAS-L (27400 seconds) was $1.21\times$ smaller than that of YARN-CS (33270 seconds), and it was similar to TIRESIAS-G (27510 seconds) and SRTF (28070 seconds).

Table 3.5: Queueing delays for DDL jobs for different solutions.

	Average	Median	95th
YARN-CS	8146s	7464s	15327s
SRTF	593s	32s	3133s
TIRESIAS-G	1005s	39s	7933s
TIRESIAS-L	963s	13s	7755s

3.5.2.3 Sources of Improvements

Smaller queueing delays. TIRESIAS's scheduler can reduce the average queueing delay of all jobs (Table 3.5), especially for small and short jobs. The average queueing delay is reduced from over 8000 seconds to around 1000 seconds when comparing YARN-CS and TIRESIAS. More importantly, half of the jobs are just delayed for less than or equal to 13 (39) seconds in TIRESIAS-L (TIRESIAS-G), which is negligible compared to the median delay in YARN-CS. Note that while TIRESIAS's average queueing delay is higher than SRTF, smaller jobs actually experience similar or shorter delays.

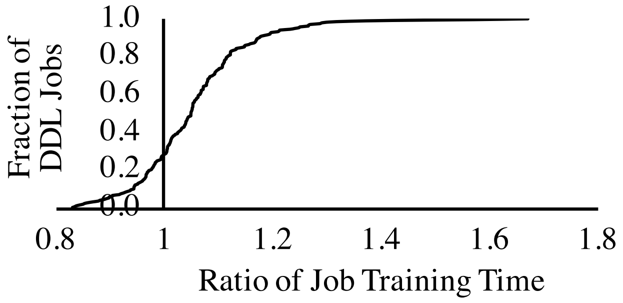


Figure 3.12: Performance improvement from job placement in TIRESIAS-L. We pick all the DDL jobs and compare their training times when TIRESIAS-L is running with and without placement.

Faster training. Albeit smaller, another source of performance improvement is TIRESIAS's job placement algorithm. To illustrate this, we rerun the experiment using TIRESIAS-L but without its profiler; i.e., jobs are randomly placed on the cluster. We compare the training time of DDL jobs in TIRESIAS-L without job profiling versus in the original

TIRESIAS-L. We use the ratio of training time in random placement to TIRESIAS-L as the factor of improvement. In Figure 3.12, large ratio means random placement slows down the training, and vice versa; single-GPU jobs are excluded from the figure. TIRESIAS-L achieves up to $1.67\times$ improvement w.r.t. random placement, because it can identify sensitive jobs and place them on minimal number of machines for better performance. Fewer than 30% of DDL jobs experience limited performance loss.

Because of the highly-skewed job distribution and the variety of model types, the major improvement comes from the job scheduling by avoiding HOL blocking of small/short jobs by the large/long ones.

3.5.2.4 Overheads

Because TIRESIAS uses preemption in its scheduling algorithm, its major overhead comes from preempting DDL jobs. The Discretized 2DAS scheduler in TIRESIAS provides *discretized* priority levels to jobs. Hence, two cases trigger job preemptions in TIRESIAS: job arrivals/promotions and demotions that change the priority queue of a job. In our experiments, TIRESIAS-L spent 13724 seconds performing 221 preemptions; TIRESIAS-G triggered 297 preemptions with 17425 seconds overhead in total. There are more preemptions in TIRESIAS-G because jobs in the same queue are sorted based on their Gittins index value at every event. In TIRESIAS-L, jobs will not be re-sorted because of FIFO ordering.

In contrast, job priorities in SRTF are *continuous*. Whenever short jobs come in, jobs that with longer remaining time (lower priorities) may be preempted due to lack of resources. Overall, SRTF spent 18057 seconds for 316 preemptions.

Note that the exact overhead of each preemption depends on the specific job and cluster conditions.

3.5.3 TIRESIAS in Trace-Driven Simulations

Here we evaluate TIRESIAS's performance on the Microsoft job trace. We compare it against YARN-CS, SRTF, and Best-effort, where Best-effort is defined as YARN-CS but without HOL blocking – i.e., it allows small jobs to jump in front of large jobs that do not have enough available GPUs.

3.5.3.1 Simulator Fidelity

We replayed the workload used in our testbed experiments in the simulator to verify the fidelity of our simulator. We found the simulation results to be similar to that of our testbed results – $5.11\times$ $(1.50\times)$ average (95th percentile) improvement w.r.t. YARN-CS, $0.74\times$ $(0.55\times)$ w.r.t. SRTF, and $1.01\times$ $(1.13\times)$ w.r.t. TIRESIAS-G. Because the simulator cannot capture overheads of preemption, the impact of placement, or cluster dynamics, the results are slightly different.

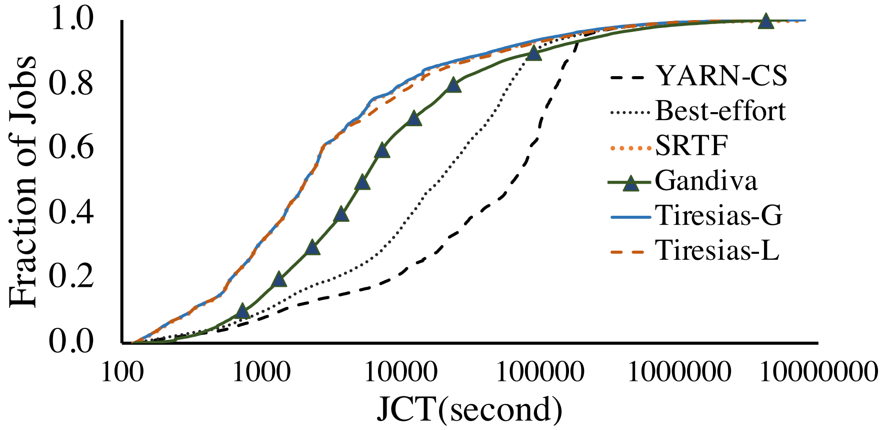


Figure 3.13: JCT distributions using different solutions in the trace-driven simulation. The x-axis is in logarithmic scale.

3.5.3.2 JCT Improvements

We then simulated the job trace from Microsoft to identify large-scale benefits of TIRESIAS (Figure 3.13). TIRESIAS-L improves the average JCT by $2.4\times$, $1.5\times$, and $2\times$ over YARN-CS, Best-effort, and Gandiva, respectively (Table 3.6). In addition, TIRESIAS-L reduces the median JCT by $30.8\times$ ($9\times$) w.r.t. YARN-CS (Best-effort). This means half of the Microsoft jobs would experience significantly shorter queueing delays using TIRESIAS. Compared to TIRESIAS-L, TIRESIAS-G has almost the same (median JCT) or slightly better (average and 95th percentile JCT) performance. More importantly, TIRESIAS performs similar to SRTF that uses complete knowledge.

Table 3.6: Improvements in JCT using TIRESIAS in simulation. Numbers are normalized by that of TIRESIAS-L.

	Average	Median	95th
YARN-CS	$2.41\times$	$30.85\times$	$1.25\times$
Best-effort	$1.50\times$	$9.03\times$	$1.08\times$
SRTF	$1.00\times$	$1.00\times$	$0.84\times$
Gandiva	$2.00\times$	$2.59\times$	$2.08\times$
TIRESIAS-G	$0.97\times$	$1.00\times$	$0.85\times$

3.5.4 Sensitivity Analysis

Here we explore TIRESIAS's sensitivity to K (number of priority queues), queue thresholds (server quantum Δ), and PROMOTEKNOB. By applying the Discretized 2D-LAS algorithm, TIRESIAS relies on K and corresponding thresholds to differentiate between jobs.

In this section, we use (K, threshold1, threshold2, ...) to represent different settings in TIRESIAS. For example, (2, 1h) means TIRESIAS has 2 priority queues and the threshold between them is 1 hour GPU time.

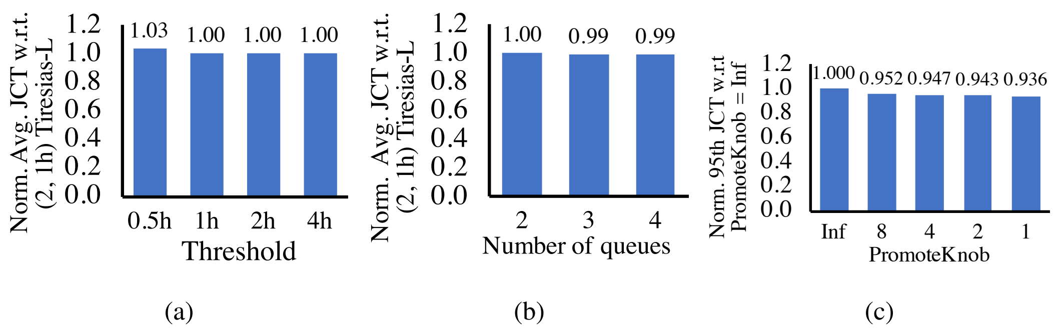


Figure 3.14: Sensitivity analysis of TIRESIAS-L. The queue settings in (b) are (2, 1h), (3, 1h, 2h), and (4, 1h, 2h, 4h) for each bar.

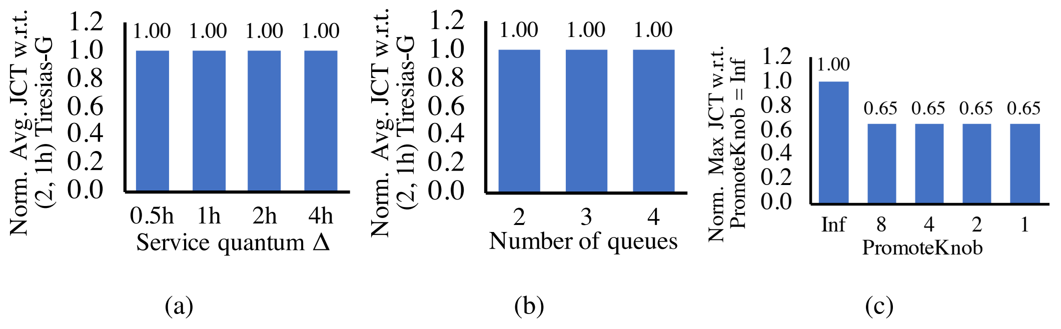


Figure 3.15: Sensitivity analysis of TIRESIAS-G. The queue settings in (b) are (2, 1h), (3, 1h, 2h), and (4, 1h, 2h, 4h) for each bar.

3.5.4.1 Impact of K (number of priority queues)

3.5.4.2 Impact of Queue Thresholds

We use TIRESIAS with K=2 and increase the threshold between the two priority queues (Figure 3.14a and 3.15a). We observe that (2, 0.5h) is slightly worse than others who have larger thresholds in terms of the average JCT in TIRESIAS-L. When the threshold is larger than or equal to 1 hour, TIRESIAS-L's performance almost does not change. For TIRESIAS-G, different Δ values have almost the same performance. These are because 1h GPU time can cover more than 60% of all the jobs.

Next, we examine TIRESIAS's sensitivity to K. We evaluate TIRESIAS with K set to 2, 3 and 4, and pick the best thresholds in each of them. The number of priority queues does not significantly affect TIRESIAS (Figure 3.14b and 3.15b). The 3- and 4-queue TIRESIAS only improves the average JCT by 1% in comparison to the 2-queue TIRESIAS-L.

3.5.4.3 Impact of PROMOTEKNOB

This simulation is based on (2, 1h). We pick the initial PROMOTEKNOB as 1, and increase it by the power of 2. When PROMOTEKNOB is infinite, TIRESIAS does not promote. Smaller PROMOTEKNOB means more frequent promotions of long-delayed jobs back to the highest priority queue. For TIRESIAS-G, the maximal JCT is cut down by PROMOTEKNOB (Figure 3.15c). However this trace is not sensitive to the different value of PROMOTEKNOB. In Figure 3.14c, the 95th JCT minutely changes when we use smaller PROMOTEKNOB in TIRESIAS-L – the key reason PROMOTEKNOB has little impact for this trace is due to its heavy-tailed nature [134].

3.6 Discussion

Formal analysis. Although Discretized 2DAS has advantages in minimizing the average JCT of DL jobs, formal analyses are still needed to precisely present its applicable boundaries (in terms of cluster resources and DL jobs' requirements). This will simplify

TIRESIAS configuration in practice.

Lightweight preemption. Existing preemption primitives for DL jobs are time-consuming. To reduce the number of preemptions, TIRESIAS adopts priority discretization using MLFQ (§3.3.2.3). A better way of preempting DL jobs has been proposed in Gandiva [164]. However, that approach requires DL framework modifications. At the same time, its overhead is still non-negligible. With lightweight preemption mechanisms, many classic and efficient algorithms in network flow and CPU scheduling can be applied for DDL scheduling.

Fine-grained job placement. TIRESIAS's profile-based placement scheme coarsely tries to avoid network transfers when necessary. However, there can be interferences within the server (e.g., on the PCIe bus) when too many workers and parameter servers are collocated. Further investigations on how placement can affect job performance are required. To this end, possible approaches include topology-aware schemes [43] and fine-grained placement of computational graphs in DL jobs [126].

3.7 Related Work

Cluster Managers and Schedulers. There are numerous existing resource managers and schedulers for CPU-based clusters for heterogeneous workloads [78, 94, 159, 161, 174, 86, 102, 81, 158, 87] or for traditional machine learning jobs [97, 157, 171]. As explained in Section 3.1, these frameworks are not designed to handle the unique characteristics of DDL jobs – e.g., all-or-nothing task scheduling, and unpredictable job duration and resource requirements – running on GPU clusters.

Resource Management in DDL Clusters. Optimus [140] is an online resource scheduler for DDL jobs on GPU clusters. It builds resource-performance model on the fly and dynamically adjusts resource allocation and job placement for minimizing the JCT. It is complementary to TIRESIAS in terms of job placement, because the latter focuses on the efficiency of the initial job placement based on job characteristics, while the former performs online adjustment according to a job's realtime status. However, Optimus assumes that the remaining time of a DL job is predictable, which is not always true in practice (§3.2.2). TIRESIAS can schedule jobs without any or with partial prior knowledge, and it does not rely on such assumptions. Gandiva [164] is a resource manager for GPU clusters that gets rid of the HOL blocking via GPU time sharing. However, the time-slicing scheduling approach in Gandiva brings limited improvement in terms of the average JCT. Themis [120], $Gandiva_{fair}$ [54] and AlloX [108] are resource managers that focus on how to fairly share the DL cluster among multiple jobs or users. Themis proposes a new fairness metrics for DL training jobs and allows trading off fairness for efficiency without affecting long-term fairness. $Gandiva_{fair}$ achieves fairness and efficiency in DL clusters with heterogenous GPUs. It provides performance isolation between users and uses a novel resource trading mechanism for maximizing cluster efficiency. AlloX manages hybrid computing resources (e.g., CPU and GPU) that are interchangeable for DL training jobs. It proposes a dynamic fair allocation algorithm that achieves fairness among users and prevents starvation. AntMan [165] co-designs cluster manager and DL frameworks. It introduces dynamic scaling mechanisms in DL frameworks by utilizing the spare GPU resources. AFS [98] is an elastic resource sharing scheme for DL clusters. It could balance resource efficiency and short job prioritization for average JCT minimization.

Resource Management with Partial or No Information. To the best of our knowledge, TIRESIAS is the first cluster scheduler for DDL training jobs that minimizes the average JCT with partial or no information. While similar ideas exist in networking [50, 59] and CPU scheduling [48, 61], GPU clusters and DDL jobs provide unique challenges with high preemption overheads and all-or-nothing scheduling. There exist all-or-nothing gang schedulers for CPU, but they are not information-agnostic. While fair schedulers do not require prior knowledge [13, 14, 169], they cannot minimize the average JCT. Similar to the Gittins index policy, Shortest Expected Remaining Processing Time (SERPT) [151] just needs partial knowledge of job durations. However, the Gittins index policy is proven to be better because it prioritizes a larger number of potentially shorter jobs [151].

3.8 Conclusion

TIRESIAS is a GPU cluster resource manager that minimizes distributed deep learning (DDL) jobs' completion times with partial or no a priori knowledge. It does not rely on any intermediate DL algorithm states (e.g., training loss values) or framework specifics (e.g., tensors-to-parameter server mapping). The key idea in TIRESIAS is the 2DAS scheduling framework that has two scheduling algorithms (*Discretized 2D-LAS* and *Discretized 2D-Gittins index*). They can respectively minimize the average JCT with no and partial prior knowledge. Additionally, TIRESIAS's profile-based job placement scheme can maintain the resource (GPU) utilization of cluster without hurting job performance. Compared to a production solution (Apache YARN's Capacity Scheduler) and a state-of-the-art DDL cluster scheduler (Gandiva), TIRESIAS shows significant improvements in the average JCT.

CHAPTER IV

AutoPS: Elastic Model Aggregation with Parameter Service

4.1 Introduction

DL has become a major driving force in many application domains, including image classification, speech recognition, and machine translation [93, 95, 68]. With increasing model complexity and large training datasets, many DL models are trained in a distributed manner. Training distributed deep learning (DDL) on large *shared* clusters is becoming prevalent [113, 100, 3, 4, 10] to meet the ever-growing demands.

Although GPU scheduling has so far received the most attention in the context of DL clusters [120, 89, 164, 140] and rightly so, we observe that the impact of traditional cluster resources such as CPU can be significant (§4.2). This is especially true for DDL training jobs, where parameter servers run on traditional virtual machines (VMs) or containers. When running those jobs, up to 80% of the allocated CPUs may be wasted due to the bursty and periodic nature of model aggregation. Notice that, CPUs are not free to use. Renting one CPU core in the cloud takes around \$900/yr [75]. Given the increasing number of DDL training jobs, users spend a substantial amount of money on the CPUs that are left unused.

The root cause of this CPU underutilization is forcing resource management techniques from big data clusters onto emerging DDL jobs. A VM or container allocated to a parameter server has a *fixed* amount of CPU resource, which is exclusive (non-sharable) and provisioned for peak usage. However, a parameter server usually cannot keep its assigned CPU saturated – CPU consumption of DDL training is inherently bursty. Parameter servers of a DDL job must idly wait for the workers' model updates that are generated layer by layer according to the progress of back-propagation.

To remedy this problem, we propose Parameter Service, an elastic model aggregation framework for DDL training that aims to improve overall CPU utilization without sacrificing training performance. Unlike prior work on model aggregation [30, 35] where parameter servers are assigned to individual jobs, Parameter Service *decouples* model aggregation from training and exposes a shared model aggregation service to all training jobs for better CPU utilization. Parameter Service is a middle layer between the DL framework and the low-level infrastructure. Therefore, it is transparent to the users (and applications) and only requires a few modifications to the DL framework.

The crux of the problem is answering: *How to efficiently share CPU resources for model aggregation among DDL jobs without degrading their performance?* Parameter Service relies on two key knobs to address this question: *dynamic workload assignment* and *elastic resource management*. It can flexibly pack the model aggregations from the same or different jobs onto a single server to fill its idle CPU cycles so as to avoid resource wastage. When any workload change occurs (e.g., job arrivals and/or exits), Parameter Service can dynamically update the assignments to enhance resource efficiency and preserve job performance. In addition, Parameter Service manages CPU resource elastically at the server level. The number of servers for model aggregation can be seamlessly scaled up or down based on the change of load in Parameter Service.

Moreover, dynamic model aggregation management requires efficient orchestration among servers, which would otherwise elongate the execution of workers and waste their GPUs. Leveraging two unique characteristics of DDL training, Parameter Service can flexibly re-assign model aggregations with negligible time overhead. First, model aggregations happen at the tensor level with no data dependencies between them. Thus, Parameter Service can reassign a single model aggregation without interrupting the entire DDL job. Second, there are not many constraints (e.g., data locality and special resource requirements) that prevent a server from accepting the re-assigned tasks. Once a decision is made, the master copy of tensor data needs to be migrated from the original server to the new one. Because DDL training is done iteratively with a fixed execution pattern, Parameter Service hides the time overhead by only performing data migration when the job is in the training stage at the worker side.

We propose a heuristic for assignment of model aggregations and a simple feedback-based scheme for resource scaling. Relying on the iterative nature of DDL training, our assignment scheme finds proper servers and gets cyclic execution slots for model aggregations. Incorporating with model aggregation assignment, our resource scaling mechanism balances between resource utilization and job performance. We have implemented Parameter Service in a system called AUTOPS and deployed it on a real DL cluster. We evaluated it using Apache MXNet [25] with multiple state-of-the-art DL models. AUTOPS reduces up to 75% of CPU resource from workload packing, with very-limited performance impact on the training jobs.

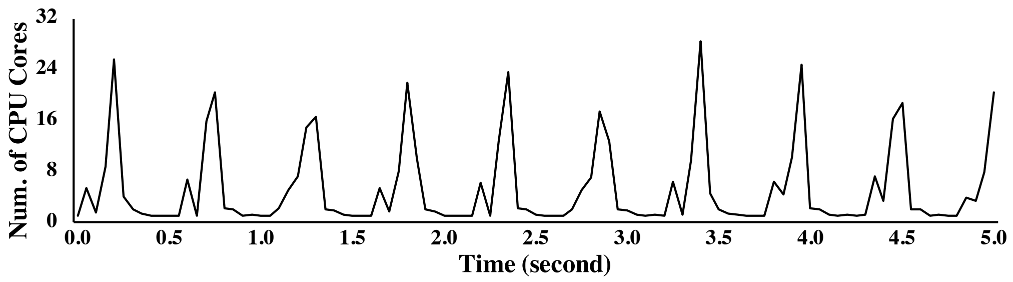
Overall, we make the following contributions in this paper:

- Parameter Service decouples model aggregation from DDL training and elevates it as a shared cluster-wide service. This makes it easier for users to run DDL training without maintaining parameter servers for each DDL job.
- Parameter Service resolves the mismatch between DDL training and the infrastructure. It improves CPU utilization by dynamic workload assignment and elastic resource management.
- Parameter Service is completely transparent to users and requires trivial modifications in the popular DL frameworks.

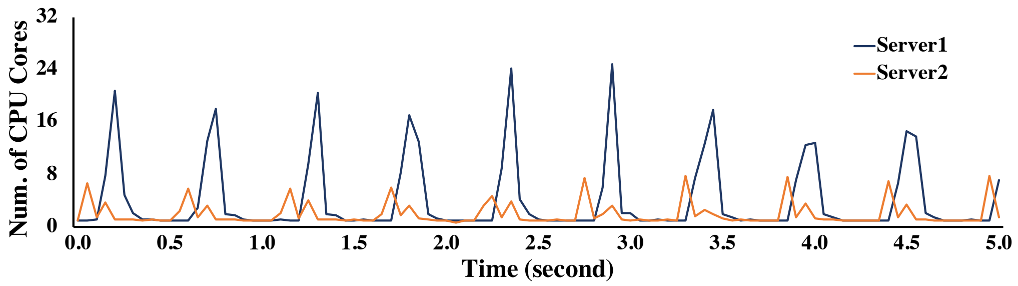
4.2 Background and Motivation

In this work, we focus on the classic *data-parallel* DDL training (§1.1.2.1) using the Parameter Server (PS) architecture(Figure 1.3a), where multiple workers work on their local copies of the DL model in parallel and the training dataset is partitioned across all the workers.

Static Parameter Assignment. For performance reasons, a single DDL job may have multiple PS instances, each of which host different parts (i.e., parameter) of the model (Figure 1.3a). In current PS design [112, 141], the assignment of model parameters is often *static* once the job starts. Thus, the workload on each PS instance will not be changed as long as the job is running.



(a) CPU usage in VGG19 (**1**s-2w) training.



(b) CPU usage in VGG19 (**2**s-2w) training.

Figure 4.1: CPU usage of model aggregation in VGG19 training. VGG19 is trained on MXNet with 2 different distributed settings: 1 PS server and 2 workers, 2 PS servers and 2 workers. Servers and workers are distributed to different machines. "1s-2w" means the job has 1 PS server and 2 workers.

4.2.1 CPU Underutilization

Most DDL training jobs run in shared clusters to attain cost-effectiveness. Despite its benefits, there is a crucial mismatch between DDL applications and the resource management software of the infrastructure. Many CPU cores, that are statically assigned to parameters server containers or VMs, are wasted due to the bursty model aggregations in DDL training.

In each training iteration, the aggregation of a tensor cannot start until the completion of backward computation on that tensor. Due to this dependency, the CPU resource reserved for model aggregation at server side is mostly idle when the job is in the *forward-and-backward* stage (Figure 1.3b). Besides, model aggregation is performed at tensor-scale. There will be a spike of CPU usage when a tensor is ready to be updated. Therefore, the

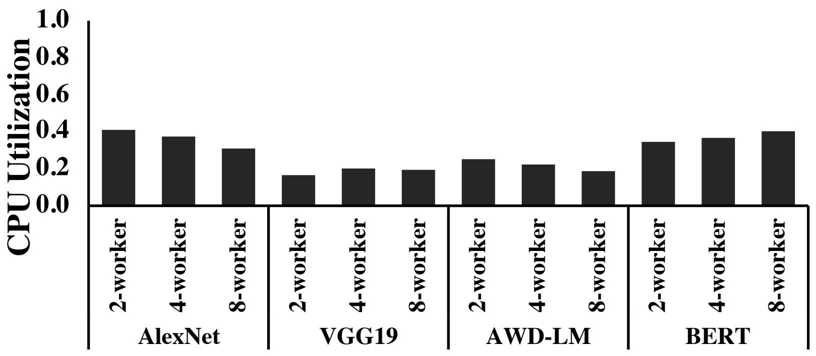


Figure 4.2: Average CPU utilization of model aggregation. Each model is trained on MXNet with 1 PS server and different number of workers.

CPU usage of model aggregation is a combination of sharp spikes and idle cycles (Figure 4.1a). In a shared cluster, CPU resource are assigned through the abstraction of VM or container for isolation. To achieve good performance, users often oversubscribe the CPU resource of their virtual machines (or containers) to satisfy the peak demand. Due to the mismatch between the dynamic CPU usage and the static CPU allocation, CPUs reserved for model aggregation remain underutilized.

Figure 4.2 shows the average CPU utilization of model aggregation when training different models. The number of reserved CPU cores for each job's PS server is the same as its peak usage. Since there is only 1 PS server in each job, the CPU consumption of model aggregation is concentrated at the single PS server. Among all the jobs, more than a half of the CPU resource are left unused. For VGG19 (1s-2w), the average CPU utilization of its server instance is only 16%.

This issue will get worse when multiple PS servers are used in a single training job. Since the workload of model aggregation is divided among those PS servers, the corresponding CPU consumption is also split (Figure 4.1). Users need to reserve plenty of CPU cores for each PS server without violating the spikes on each of them.

4.2.2 Opportunities Brought by DDL training

DDL training has unique characteristics that create opportunities to resolve the mismatch with infrastructure.

Tensor-Based Model Structure. Although tensors are layered in the model and have dependencies with their neighbors in *forward-and-backward* computation, they are independent of each other when they are getting aggregated. Each tensor can be managed individually at the server side. Based on this feature, it is possible to apply *fine-grained and dynamic workload management*. Model aggregations can be independently mapped to the proper CPU slots for execution. The CPU utilization could be improved by packing more model aggregations from multiple jobs in a single server instance.

Iterative Training. To handle the mismatch mentioned above, the information of the training job, especially model aggregation, is required. Relying on the iterative feature, the runtime information (e.g., CPU consumption) of the training job measured in the previous iterations can be used as the input for making long-term decisions in workload management.

4.3 Parameter Service

Parameter Service is an elastic model aggregation framework for DDL training. It aims at enhancing CPU utilization for model aggregation without sacrificing job performance. In this section, we first present the overview of Parameter Service followed by how to migrate tensor data without negligible time overhead. We then illustrate how model aggregations are managed by Parameter Service which includes the schemes of workload assignment and resource scaling.

4.3.1 System Overview

To achieve the aforementioned goals, Parameter Service introduces two features, *dynamic workload assignment* and *elastic resource allocation*, to the traditional parameter server-based approach. When the total load in Parameter Service changes (e.g., job arrival and completion), workload reassignment might be triggered if Parameter Service finds any CPU slots that fit better for some existing model aggregations. Meanwhile, incorporating with workload (re)assignment, the number of model aggregation servers will be scaled up or down based on demands.

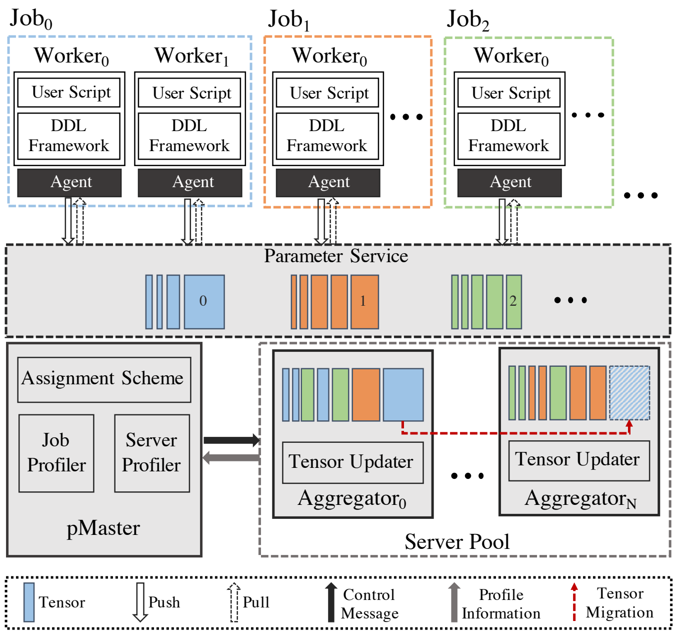


Figure 4.3: Parameter Service architecture. Agent is loaded under the DL framework layer at each worker. There is a pMaster in the cluster that manages the pool of Aggregator. Each Aggregator is placed on an individual machine.

With Parameter Service, model aggregation is decoupled from individual training jobs, whereby their workers only need to submit the model aggregation requests to Parameter Service through the unique interface and wait for the response, without worrying about where the requests are handled and how much of resource to be allocated (Figure 4.3). In the backend, Parameter Service carefully assigns those requests to the available servers. In the Parameter Service design, model aggregations are not grouped by training job any longer; each one is independent and can share the aggregation server with the ones from other training jobs.

Parameter Service has three components (Figure 4.3):

1. ***pMaster*** is a centralized manager. It has a job profiler and a server profiler that keep monitoring the status of training jobs and resource availability (i.e., CPU) of each Aggregator, respectively. When anything is changed, it will adjust the workload assignment and resource allocation accordingly.

2. ***Aggregator*** maintains the model tensors of different jobs and handles their aggregation requests. There is a pool of Aggregators managed by pMaster. pMaster elastically adjusts the number of its Aggregator according to the change of total load. Any two Aggregators can migrate the workload from one to the other, when pMaster reassigns the workload between them.

3. ***Agent*** is the interface that Parameter Service exposes to the workers. Each one works for a single worker. It maintains a mapping table that keeps the information of Aggregator for tensors in the job. When a model aggregation request comes, it will forward the request from its worker to the destination by checking the table.

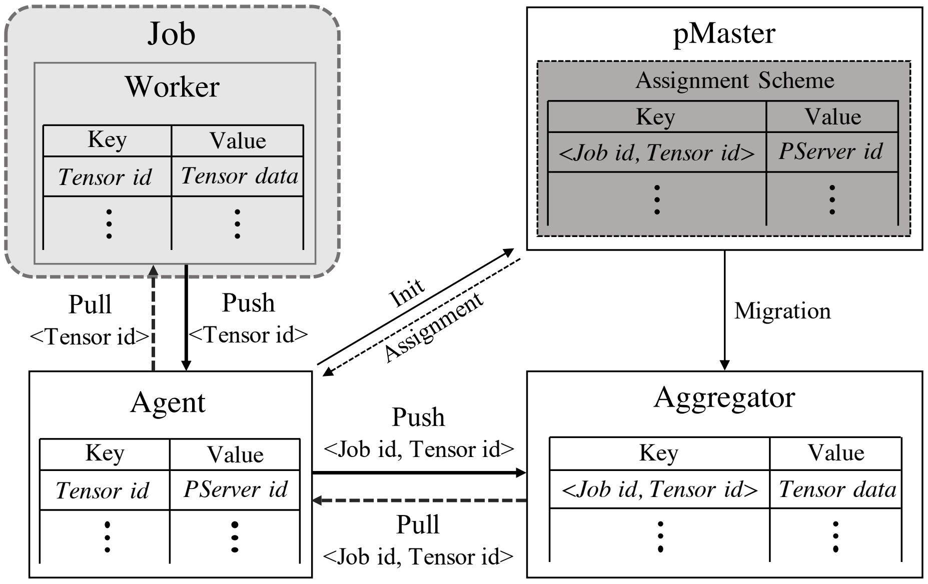


Figure 4.4: Interactions among worker and Parameter Service components. Model aggregation requests in Parameter Service are identified by the combination of job ID and tensor ID. Agent sends *Init* to Parameter Service when a new tensor arrives. Aggregator receives *Migration* from Parameter Service when one of its tensor is reassigned.

4.3.1.1 Decoupled Model Aggregation

To dynamically assign workload to the proper CPU slots, Parameter Service decouples model aggregations from their training jobs.

Originally, each training job maintains a group of parameter servers (PS) of its own. As mentioned in §4.2, the assignment of model aggregation is static once the job starts. Model aggregation requests use a *Key-Value* format. The key field is filled with the tensor ID, so that the request can be easily identified at both worker and PS side.

As a major mechanism in Parameter Service, the function of workload assignment is moved from individual training job to pMaster (Figure 4.4). In order to avoid application modification, Parameter Service keeps the *Key-Value* interface for model aggregation. For each worker, there is one Agent that has a mapping table for tensor assignment. It can assist the worker to figure out where the requests should be forwarded. When an aggregation request of a new tensor arrives at Agent, it will send the initial request to pMaster for assignment. Other than tensor ID, the job information (i.e., job ID) also has to be kept for each model aggregation request in Parameter Service, since an Aggregator might be shared by multiple training jobs. Therefore, the key field of aggregation request is transferred to be a pair of job ID and tensor ID by Agent before sending to the destination. Aggregator uses the updated key to identify model aggregation requests.

4.3.2 Tensor Migration

There is one challenge: the aggregation servers keep the master copy (latest version) of tensors for model updating (Figure 1.3). Blocked by this data dependency, model aggregations cannot be freely reassigned to different servers.

In the current PS systems [30, 35], reassigning a single model aggregation task will interrupt the entire training job. It needs to pause the training process, checkpoint the model parameters, and resume the job with the new assignment, which brings tens of seconds overhead to the training process. In Parameter Service, we propose a deep-learning-specific migration mechanism that migrates tensor data between the Aggregators and updates assignment information in the mapping table of each Agent. It is performed at the tensor-level and exploits the features of DDL training for reducing time overhead and ensuring data consistency.

Figure 4.5 shows the procedure of tensor migration in our design. pMaster initiates a migration request when a model aggregation needs to be reassigned. At ①, $Aggregator_{old}$ receives the migration request (*MIGRATE_INIT*) and keeps the information of the tensor and $Aggregator_{new}$. When the tensor is needed (*Pull*) by the workers at the beginning of next iteration (②), $Aggregator_{old}$ embeds the information of $Aggregator_{new}$ into the response and sends to the workers. Once workers receive the information (③), their Agents update the mapping table for the tensor. Then, workers can continue the *forward-backward* computation. Meanwhile, $Aggregator_{old}$ copies the contents (e.g., metadata, and model parameters) of the tensor (*TENSOR_COPY*) to $Aggregator_{new}$ once the response to workers completes (④). At $Aggregator_{new}$ side, it adds the incoming tensor into its tensor list (⑤). When the copy finishes (⑥), $Aggregator_{old}$ will notify pMaster (*TENSOR_COPY_DONE*) that the tensor data has arrived at $Aggregator_{new}$. After getting the gradient of the tensor in *backward* computation, workers pushes their results to $Aggregator_{new}$. Once those local results arrive (⑧), $Aggregator_{new}$ also notifies pMaster (*WORKER_DONE*) that the workers have the updated tensor assignment. The migration request completes after pMaster receives notifications from both $Aggregator_{old}$ and $Aggregator_{new}$.

Negligible Time Overhead. In model aggregation, the master copy of tensor at the Aggregator side is only needed when it is being updated. There is a large time window (from the completion of the last *Pull* to the start of *Update*) in each iteration that the tensor copy hosted by Aggregator is not accessed (Figure 1.3b). The actual migration of tensor data is performed within this window to hide its time overhead as much as possible. In many cases, a tensor migration exposes negligible or even zero time overhead to the training job.

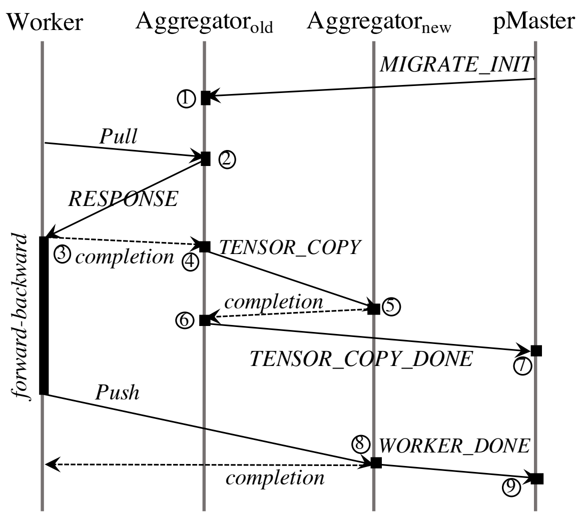


Figure 4.5: Procedure of tensor migration in Parameter Service. $Aggregator_{old}$ denotes the old Aggregator, and $Aggregator_{new}$ means the new one. *Pull* and *Push* are the original messages in model aggregation. The *completion* notifications of data transmission are from the network transport layer.

Data Consistency. There are two data consistencies that should be guaranteed during tensor migration: (1) the mapping table among Agents, and (2) the master copy of the migrating tensor between $Aggregator_{old}$ and $Aggregator_{new}$. Parameter Service merges the procedure of tensor migration into the iterative training procedure. The information of $Aggregator_{new}$ is added into the response message of *Pull* request. The corresponding mapping table of each Agent can be updated, as long as its worker receives the updated tensor. Thus, the consistency of the mapping table among Agents is guaranteed. Besides, $Aggregator_{new}$ will not execute model update on the tensor that is under migration until the tensor data is completely copied (*TENSOR_COPY*) from $Aggregator_{old}$. This insures the right version of tensor at $Aggregator_{new}$ is used in the following model aggregation.

4.3.3 Model Aggregation Management

The core of Parameter Service lies in its schemes of model aggregation assignment and resource scaling that aim at improving CPU utilization without losing job performance.

When assigning model aggregations, Parameter Service has to carefully balance the trade-off between job performance and resource utilization. Allocating too many Aggregators is good for the jobs because of less resource contention but sacrifices resource utilization; and vice versa. One extreme is the current parameter server solution that allocates individual parameter servers for each training job. When an Aggregator has surplus capacity, Parameter Service will opportunistically pack more model aggregations on it for resource efficiency. It follows the principle that a training job should not lose its performance when other jobs share Aggregators with it.

Parameter Service can elastically change the amount of resource (i.e., the number of Aggregators) according to the total load of model aggregation in the cluster. There are two events that may trigger Aggregator scaling: (1) new training job arrival; (2) existing job exit. When a new training job arrives, Parameter Service will assign its model aggregations onto the existing Aggregators as much as possible. If they do not fit, new Aggregators will be allocated for the extra workload. For job exit, Parameter Service will return the empty Aggregators back to the cluster manager to avoid wastage. Moreover, it explores the opportunity of freeing the least-loaded Aggregators by trying to reassign the model aggregations to other Aggregators.

4.3.3.1 Profiler

Parameter Service requires the information of training jobs and infrastructure when managing workload and resource under it. For a new job, Parameter Service needs to know its characteristics including the iteration duration and the detailed resource consumption of every tensor in model aggregation. Because of the iterative nature of DDL training, these characteristics can be profiled through several trial iterations, and then used for future training. When a job arrives, Parameter Service will temporarily allocate a new group of Aggregators to it for profiling. The number of temporary Aggregators is the same as the number of parameter servers in the job settings which are configured by its user. Once a job starts, Parameter Service keeps periodically calibrating those job information. Similarly, Parameter Service periodically monitors the load and CPU usage of each Aggregator.

4.3.3.2 Model Aggregation Assignment

After collecting the characteristics of a new training job, Parameter Service needs to assign the model aggregations from the temporary Aggregators to the stable ones and allocate new ones if needed. The objectives here includes minimizing the total number of Aggregator for resource efficiency, and balancing the load of each Aggregator for less resource contention among model aggregations

It is infeasible to tackle this assignment problem at scale in an online manner. When many DDL jobs are served by Parameter Service, the large number of tensors (parameters) in DL models and fast training speed generate high volume of model aggregation requests[1] per second, which can easily overwhelm the scheduler and lead to non-negligible queuing

[1]Each model aggregation task has one request per iteration.

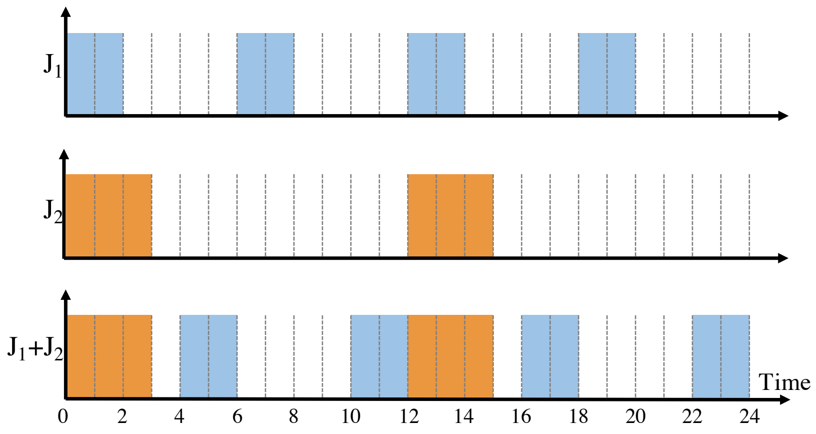


Figure 4.6: Toy examples of cyclic execution of Aggregator. In the two figures at the top, the Aggregator only serves one job (J_1 or J_2). In the bottom one, it serves both J_1 and J_2. The iteration duration of J_1 (J_2) is 6 (12) units of time; its model aggregation takes 2 (3) units of time.

delay. Based on the periodicity of model aggregation workload, we propose an offline approach that gets fixed and cyclic execution slots for model aggregations.

Cyclic Execution. Aggregator has an execution cycle that covers the execution slots of model aggregation tasks assigned to it. The execution cycle is determined by the model aggregations on it and is updated when the assignments change. When Aggregator serves model aggregations from one job, its execution cycle is simply the job's (also those model aggregations') iteration duration. When model aggregations from multiple jobs are packed together, Aggregator picks the largest iteration duration among those jobs as its execution cycle. With that, all the model aggregations can be executed within a single cycle. And the jobs with smaller iteration duration may gets executed for multiple iterations. In the toy examples of Figure 4.6, the execution cycle of Aggregator is 6 (12) units of time when only tasks of J_1 (J_2) are assigned to it. If the tasks of J_1 and J_2 are packed together, Aggregator will have an execution cycle for 12 units of time. The model aggregations from J_1 will be executed twice within one cycle.

With this cyclic execution design, assigning a model aggregation task could change the execution cycle of Aggregator, which affects the execution of existing tasks. For example, there is an Aggregator serving a model aggregation task whose execution time (iteration duration) is 1 (5) unit of time. If the task of J_2 in Figure 4.6 is assigned to it, then its execution cycle will be 12 units of time. Accordingly, the iteration duration of the existing task will be 6 units of time, since the task can run twice in one cycle. Theoretically, the job of the existing task may lose 17% of its training speed.

Table 4.1: Notations in the IP problem and assignment scheme.

Notation	Description
C_n	Execution cycle of Aggregator n
C_n^{est}	Estimated C_n
D_j	Profiled iteration duration of job j
d_j	Current iteration duration of job j
e_t	the execution (CPU) time of task t
F_n^{est}	Estimated free CPU slots on Aggregation n
W_n	Total execution time of tasks on Aggregation n
L_n	Performance (i.e., training speed) loss of job j
$\mathbb{N}$	Set of allocated Aggregators
$\mathbb{J}$	Set of training jobs
$\mathbb{J}_n$	Set of jobs that have tasks on Aggregator n
$\mathbb{T}_j^n$	Set of tasks on Aggregator n that belong to job j

With the purpose of eliminating potential performance loss, the assignment problem can be expressed as an integer programming problem (IP) with the objective of minimizing the performance loss of all job. The variable in this problem is p_{tn}. It is a binary variable, and indicates whether model aggregation task t is assigned to Aggregator n or not. The objective function is to minimize the maximal performance loss among all jobs, which is expressed as:

$$\text{minimize} \quad \max_{j \in \mathbb{J}} \{L_j\}$$

There are two constraints:

$$\sum_{n \in \mathbb{N}} p_{tn} = 1, \quad \forall_{j \in \mathbb{J}, t \in j} \tag{IV..1}$$

$$W_n \leq C_n, \quad \forall_{n \in \mathbb{N}} \tag{IV..2}$$

with the following definitions:

$$\begin{aligned} C_n &= \max_{j \in \mathbb{J}, t \in j} (p_{tn} \cdot D_j) \\ d_j &= \max_{t \in j, n \in \mathbb{N}} \Big(\frac{C_n}{\left\lfloor \frac{C_n}{D_j} \right\rfloor} \cdot p_{tn}\Big) \\ W_n &= \sum_{j \in \mathbb{J}} \sum_{t \in j} \Big(p_{tn} \cdot e_t \cdot \left\lfloor \frac{C_n}{d_j} \right\rfloor\Big) \\ L_j &= \frac{d_j - D_j}{d_j} \end{aligned}$$

The first constraint (IV..1) means each model aggregation can only be assigned to a single Aggregator. The second one (IV..2) requires that Aggregators should not be overloaded within each execution cycle. Due to the non-linear constraints and objective function, the problem is NP-hard and is infeasible to solve.

Here, we introduce a heuristic-based solution (Pseudocode 3). The first step of our scheme is to estimate the performance impact to the co-located jobs on each Aggregator (Line 1 - 10). Assuming the new task (t) is assigned to Aggregator n, it updates the execution cycle (C_n^{est}) of n (Line 2), and then, the iteration duration (D_j^{est}) of jobs on n (Line 4). When any job's (estimated) performance loss exceeds the predefined threshold (`LossLimit`, default is 0.1), Aggregator n will be remove from the list of assignment destination (Line 7). After examining all the Aggregators, if no Aggregator remains, the scheme will allocate a new one (Line 12) and assign the task there (Line 13). Meanwhile, how many free CPU slots (F_n^{est}) on each Aggregator is calculated with the updated execution cycle and task execution (Line 9). Among the qualified Aggregators, our scheme assigns the new task to the best-fit one who has sufficient but the least number of free CPU slots. (Line 16 - 21). In the end, if no one has enough free CPU to fit the new task, a new Aggregator will be allocated (Line 22 - 23).

After assigning model aggregations to Aggregators, Parameter Service monitors the performance (i.e., training speed) of training jobs and compares with their standalone performance which is profiled at the beginning. If there is any performance loss that exceeds the threshold (`LossLimit`), the new assignments will be reverted.

Handling Outliers in Cyclic Execution. Due to many random reasons (e.g., cache misses, and network variations), workers in a DDL training job may become transient stragglers [96, 66, 92], which makes some model aggregation requests miss their execution slots in the execution cycle. Parameter Service handles those delayed outliers in two different ways.

Pseudocode 3 Model Aggregation Assignment Scheme

```
Input  $T$ is the set of model aggregation tasks (tensors)
       $e_t$ is the execution (CPU) time of $t$
       $\mathbb{N}$ is the set of allocated Aggregators
 1: for all Aggregator $n \in \mathbb{N}$ do
 2:   $C_n^{est} \leftarrow \max(C_n, D_k)$
 3:   for all job $j \in \mathbb{J}_n$ do
 4:     $d_j \leftarrow \max(D_j, \frac{C_n^{est}}{\left\lfloor \frac{C_n^{est}}{D_j} \right\rfloor})$
 5:   end for
 6:   if $\frac{d_j - D_j}{d_j} \geq$ LossLimit then
 7:     $\mathbb{N} \leftarrow \mathbb{N} \setminus n$, and skip
 8:   end if
 9:   $F_n^{est} \leftarrow C_n^{est} - \sum_{j \in \mathbb{J}_n} (\left\lfloor \frac{C_n^{new}}{d_j} \right\rfloor \times \sum_{i \in \mathbb{T}_j^n} e_i)$
10: end for
11: if $\mathbb{N}$ is $\emptyset$ then
12:   Allocate Aggregator $s$
13:   $\mathbb{T}_k^s \leftarrow t$, $\mathbb{N} \leftarrow \mathbb{N} \cap s$
14:   return $s$ and $\mathbb{N}$
15: end if
16: for Aggregator $n \in \mathbb{N}$ in descending order do
17:   if $F_n^{est} \geq e_t$ and is the best fit then
18:     $\mathbb{T}_k^n \leftarrow \mathbb{T}_k^n \cap t$
19:     return $n$ and $\mathbb{N}$
20:   end if
21: end for
22: Allocate Aggregator $s$
23: $\mathbb{T}_k^s \leftarrow t$, $\mathbb{N} \leftarrow \mathbb{N} \cap s$
24: return $s$ and $\mathbb{N}$
```

When a request arrives late, Aggregator will check whether it has sufficient CPU slots for this delayed request after reserving enough slots for the remaining scheduled requests in current execution cycle. If so, then the outlier will get executed in this cycle. Otherwise, the request will be postponed to the next execution cycle, so that the co-located model aggregations are not affected by this outlier. In worst case, the affected job could be delayed by one iteration.

4.3.3.3 Aggregator Scaling

Other than the model aggregation assignment scheme, Parameter Service needs to scale up or down the number of Aggregators to balance the tradeoff between CPU utilization and job performance. Aggregator scaling can be triggered by two events: job arrival and exit.

Job Arrival. When a new job arrives, all of its model aggregations will be assigned by the scheme (§4.3.3.2). After that, if its performance is worse than the standalone one, Parameter Service will add a new Aggregator and re-assign the entire job. This procedure will repeat until the performance loss of job is within the threshold (`LossLimit`).

Job Exit. Parameter Service opportunistically recycles some light-loaded Aggregators when there are Aggregators released because of job exit. Starting from the least-loaded Aggregator, Parameter Service reassigns its workload to other Aggregators *without* new allocations allowed. If it succeeds, Parameter Service will recycle that Aggregator and repeat the procedure on the next least-loaded one.

4.3.3.4 Aggregator Cluster

It is a common issue that having a single centralized resource manager (i.e., pMaster) may hurt the performance of system at scale. In Parameter Service, assigning one model aggregation needs pMaster to scan all available Aggregators in the pool. One training job may have hundreds or even thousands of model aggregations (i.e., tensors) in its model. Assuming Parameter Service gets deployed in a large-scale cluster, the time complexity of assigning one training job will exponentially increase with large numbers of Aggregators

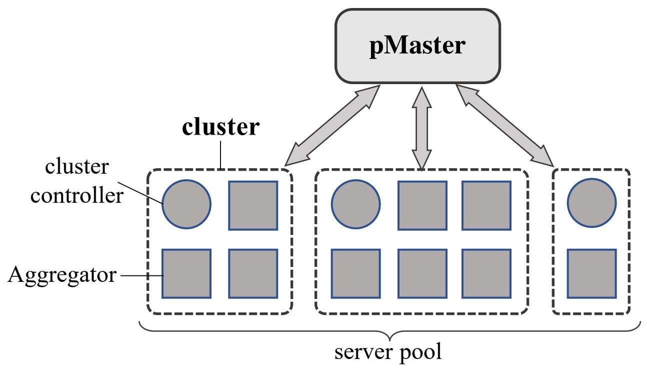


Figure 4.7: Overview of Aggregator cluster.

and model aggregations. Additionally, it could take even longer if there are new Aggregator allocations triggered in the assignments. Therefore, pMaster can be easily overwhelmed by a burst of job arrivals. Moreover, on-demand resource scaling in Parameter Service could affect the execution of running jobs through workload reassignments. A sequence of job events may thrash the workload assignments in Parameter Service, which makes it hard to get stable execution for the running jobs.

To handle this issue in Parameter Service, we apply the idea of server (i.e., Aggregator) cluster that the pool of Aggregators is split into multiple independent clusters (Figure 4.7). Each cluster has a controller who is in charge of managing the Aggregator resource in its cluster. All cluster controllers are under the management of pMaster. Workload assignment is split into two steps. A new job is firstly forwarded from pMaster to an appropriate cluster controller, then the cluster controller assigns model aggregations of the job to its Aggregators. Therefore, one job gets Aggregator resource for model aggregation from a single cluster. Each cluster works independently. There is no cross-cluster interactions.

Why Aggregator Cluster? Splitting the pool of Aggregators into multiple clusters brings the following benefits in mitigating the scalability issue. First, it is the cluster controller that allocates Aggregator resource to model aggregations. The number of assignment destinations in a cluster is much fewer than the total number of Aggregators in Parameter Service. Second, model aggregations of a job get the Aggregator resource from a single cluster. Only the jobs that are served by the same Aggregator cluster may be affected by a job event (arrival or exit). The impact of workload reassignments is limited within a single cluster. Moreover, multiple cluster controllers could perform workload assignments in parallel when there are multiple new job arrivals.

With Aggregator cluster design, Parameter Service assigns model aggregations of new jobs in two steps. First, pMaster decides which cluster should be chosen for placing the new job. pMaster keeps track of the remaining free CPU resource of each Aggregator cluster. According to the profiled job information, specially total CPU consumption of the job, pMaster selects the best-fit cluster, who has sufficient but least amount of free CPU resource, and forwards the new job there. The complexity of this step is negligible compared to the assignment scheme in Pseudocode 3. Once the job is forwarded to a cluster, the cluster controller assigns model aggregations in this job to its Aggregators following the assignment scheme in §4.3.3.2. As discussed above, the time complexity in this step is greatly reduced because of much fewer assignment destinations. When a job exits, its cluster controller should report this event to pMaster. pMaster will update the status of the cluster on its side for future job forwarding.

Hybrid Resource Scaling. As mentioned above, on-demand resource scaling triggered by job arrivals or exits may jeopardize the execution of running jobs when there are too many workload reassignments generated. Although periodically resource scaling could reduce such disruptions, it can not respond to the change of resource demand in real-time. Consequently, Parameter Service performs resource scaling in a hybrid manner. There is a predefined resource scaling period in Parameter Service. By default, Parameter Service adjusts the number of clusters and the amount of Aggregators in each cluster according to the resource demand measured in the latest period. To avoid starvation, on-demand Aggregator allocation is also allowed when the demand of new Aggregator is higher than a predefined threshold.

There are interactions between pMaster and cluster controllers when job events or resource scaling occurs. For a new job arrival, pMaster needs to forward the new job information to the chosen cluster controller for assigning model aggregations. When a job exit, the cluster controller sends job completion information to pMaster for bookkeeping. A cluster controller should send allocation or deallocation requests to pMaster when resource scaling is triggered by job events in it. Approval has to be received before the cluster controller can carry out the operation. The amount of those interaction messages is at the same order of magnitude of job arrival and exit rate. Therefore, these interactions will not be the bottleneck of Parameter Service.

4.3.3.5 Admission Control

To prevent system overload, Parameter Service relies on the admission control of the cluster manager. When the cluster manager cannot satisfy the resource requirements (a number of standalone parameter servers) of a new incoming job, it should not schedule the job to run. Because Parameter Service may require the same number of Aggregators to run the job in worst case.

4.4 Implementation

We have implemented the design of Parameter Service into a system named AUTOPS. AUTOPS is built on top of ps-lite [30] with about 5K lines of C++ code added.

In AUTOPS, pMaster is a daemon process that serves the entire cluster. It opens a connection address to the DDL training jobs who want to use Parameter Service. A training job can connect to pMaster through the Agents that are collocated with its workers. Agent is a implemented as a Key-Value library that is loaded by the DL framework of worker. Agent exposes the standard *Push* and *Pull* APIs to the Key-Value store layer in DL framework for model aggregation requests and responses. Since RDMA network is widely deployed in many DL clusters, our current implementation uses RDMA `ibverbs` for communication.

Control Plane. Every Agent and Aggregator has a control channel to the pMaster. Agent needs to send requests to pMaster for job registration and tensor initialization. In response, pMaster sends back the initial assignments of tensor. When re-assigning workload, pMaster sends the migration command to the Aggregator that is currently hosting the tensor, and waits completion notifications from the two related Aggregators. In addition, pMaster sends profiling commands to Agents and Aggregators when there are updates in workload assignment or Aggregator allocation.

Data Plane. There are two primary data plane activities in AUTOPS. The first one is between Agent and Aggregator for regular model aggregation. The other one is among Aggregators for tensor migration when pMaster re-assigns workload. To avoid the possible blocking between the two, Aggregator uses two individual threads.

Profiler. In each Agent and Aggregator, there is a profiling thread that monitors the status of job and infrastructure (e.g., iteration duration, network and CPU utilization, and so on) following the requirements from pMaster. Each profiling thread only collects one record of data within one profiling period which is configurable by pMaster. Therefore, the profilers are light-weight and have negligible impacts to the training jobs.

4.5 Evaluation

We have evaluated AUTOPS using both testbed experiments and trace-driven simulations, and highlight the results as follows:

- AUTOPS improves the resource (CPU) efficiency with none or negligible performance loss to the training jobs. It can reduce up to 75% of CPU servers comparing to the traditional parameter server approach.
- AUTOPS outperforms the traditional parameter server approach (up to $1.17\times$) when a single job uses each of them in standalone mode.
- Compared to the existing approach, AUTOPS cuts the time overhead of reassigning model aggregation from second-level to millisecond-level.
- The CPU saving benefits of AUTOPS hold for the large-scale cluster scenario in trace-

driven simulation.

4.5.1 Experimental Setup

Testbed. Our testbed consists of 8 GPU machines and 8 CPU machines. Every machine has 40 CPU cores, 256 GB DDR4 memory, and a 100 Gbps RDMA (InfiniBand) network adapter. Each GPU machine has 4 NVIDIA Telsa P100 GPUs with NVLink connections.

Workload. We train 4 classic and popular DL models (Table 4.2) using MXNet in the experiments. Two of them are CNN models (AlexNet [105], VGG19 [155]), which are trained with ImageNet [67] dataset. The other two are RNN models (AWD-LM [124], BERT [68]). AWD-LM is trained with WikiText-2 [123] dataset; BERT uses BookCorpus [176]. Batch size of each model is set to the maximum to fit into the GPU memory in our testbed. Jobs can have different training settings (the number of workers and parameter servers). In this chapter, "1s-2w" means the job requires 1 parameter server and 2 workers. Parameter server (and Aggregator) uses CPU machine, and worker runs on GPU machine. Each worker has 4 GPUs (from the one machine).

Table 4.2: Models and datasets in the experiments.

Model	Dataset	# of Tensors	Model Size
AlexNet	ImageNet	17	236 MB
VGG19	ImageNet	39	548 MB
AWD-LM	WikiText-2	14	396 MB
BERT	BookCorpus	393	1278 MB

Simulator. We build an event-based simulator and use a real job trace from Microsoft to evaluate AUTOPS when it gets deployed on a large scale cluster. It simulates all job events and resource scaling activities. For workload assignment, it uses the job information, specially CPU consumption and iteration duration, measured from actual runs.

Baseline. We compare AUTOPS to the classic parameter server implementation (ps-lite [30], also using RDMA network) which is used for distributed training in MXNet by default. Using ps-lite, each training job has an individual group of parameter servers for model aggregation. AUTOPS also takes the number of parameter servers of each job to profile the standalone performance of the job (§4.3.3.1).

Metric. We use the training speed (i.e., images or samples per second) to represent job performance. AUTOPS saves CPU resource through reducing the number of Aggregators allocated for model aggregation. In testbed experiments, we focus on how many CPU servers are reduced in AUTOPS comparing to ps-lite under the same scenarios, and define:

$$\text{CPU Reduction Ratio} = \frac{\text{\# of param. servers} - \text{\# of Agg.}}{\text{\# of param. servers}}$$

Larger value of this ratio means savings more CPU servers for model aggregation.

4.5.2 Evaluation Results

4.5.2.1 Single-Job Experiments

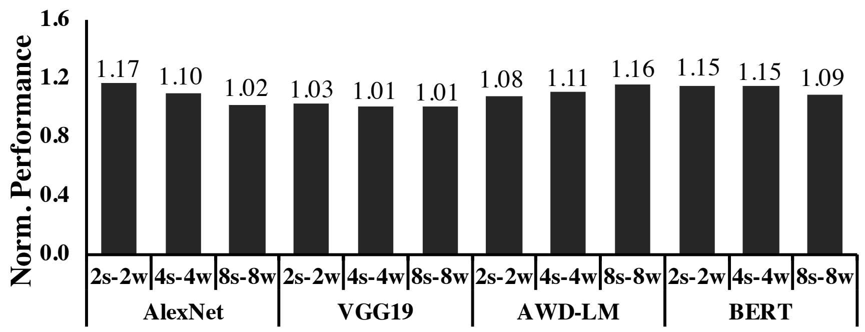


Figure 4.8: Normalized performance of single job using AUTOPS. The performance of each job is normalized by its performance when using ps-lite. For each job, it use the same number of Aggregators (AUTOPS) and parameter servers (ps-lite).

To verify the effectiveness of model aggregation function in AUTOPS, we compare the performance of jobs when they are using AUTOPS and ps-lite in standalone mode (Figure 4.8).

The performance of AUTOPS is not worse than ps-lite, which means the extra operations (e.g., extra request mappings for decoupling model aggregation, and periodic job profiling) have negligible impact on the training job. In addition, AUTOPS outperforms ps-lite by up to $1.17\times$ in some cases. These performance improvements come from the better balanced load distribution in AUTOPS, comparing to the round-robin distribution in ps-lite.

4.5.2.2 Time Overhead of Reassignment Operation

As a key knob in Parameter Service, reassigning model aggregation may be frequently triggered when there are workload changes. It could block the execution of workers and

Table 4.3: Time overhead of migrating all tensors in a model.

	AlexNet	VGG19	AWD-LM	BERT
Overhead (ms)	13.6	21.5	40.6	43.8

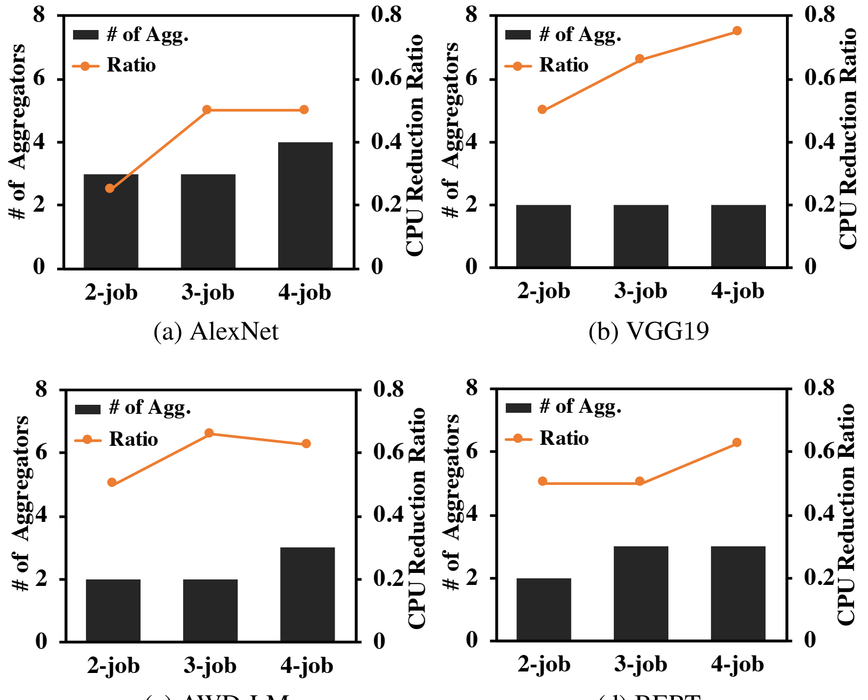
waste their GPUs when the tensor data is not totally migrated to the new Aggregator.

We measure the time overhead of reassigning model aggregation by manually reassigning all tensors in a model from one group of Aggregators to the other when the job is running, and taking the averaged value from 10 runs for each model. From the view of training jobs, they are only suspended for *tens of milliseconds* by the reassignment (Table 4.3). The actual durations of those reassignment operations are much longer, most of which are hidden under the computing time at worker side. Compared with the existing approach of reassigning model aggregation (pause, checkpoint, and resume), which halts the training job for *tens of seconds* [164, 89], the reassignment operation in AUTOPS has negligible time overhead.

AUTOPS uses protobuf [29] library to format the data before sending to the network. It introduces unavoidable overhead (e.g., data copy) by several milliseconds to each reassignment. The reassignment operation in AUTOPS could be further optimized if it directly applies remote data access feature from RDMA networks.

4.5.2.3 Multi-Job Experiments

When multiple jobs run on AUTOPS, it can opportunistically shrink the number of allocated Aggregators for resource efficiency. Here, we run multiple training jobs (with the same DL model and distributed settings) together to see how jobs' performance and CPU server allocation will be changed. Due to the limitation of machines, we can run up to 4



(a) AlexNet (b) VGG19 (c) AWD-LM (d) BERT

Figure 4.9: Number of Aggregators when multiple (2s-2w) jobs use AutoPS together. Each job requires 2 parameter servers when using ps-lite. 2-job means two jobs use AutoPS together. Jobs in the same group train the same model.

(2s-2w) jobs, or 2 (4s-4w) jobs.

CPU Reduction. Figure 4.9 shows the number of allocated Aggregators in AutoPS when multiple (2s-2w) jobs runs together. The baseline here is the total number of required parameter servers in each scenario. For example, there are 6 parameter servers needed in total when 3 (2s-2w) jobs use ps-lite for model aggregation. Comparing to ps-lite, AutoPS can save 25% to even 75% of CPU servers. The AlexNet jobs need more Aggregators than the jobs of other models. AlexNet is the only model that requires one extra Aggregator to run 2 (2s-2w) jobs. That's because of the very short iteration time of AlexNet jobs, which makes them have much higher frequency of model aggregation than others. In contrast, 2 Aggregators can serve 4 VGG19 jobs whose iteration time is much longer. Table 4.4 shows

Table 4.4: CPU reduction ratio when 2 (4s-4w) jobs use AUTOPS.

	AlexNet	VGG19	AWD-LM	BERT
Ratio	0.375	0.5	0.5	0.5

the CPU reduction ratio of AUTOPS when there are 2 (4s-4w) jobs. Same as Figure 4.9, most of the "2-job" cases can run without allocating new Aggregator.

Impact on Job Performance. In Parameter Service, the performance of training jobs should not be sacrificed for improving resource efficiency. When any workload assignment makes the job performance lower than the threshold (`LowPerf`), AUTOPS will revoke it and re-do the assignment with new Aggregator added. Figure 4.10 shows how job performance is impacted when multiple jobs uses AUTOPS. The number of allocated Aggregators of each multi-job group can be found in Figure 4.9. The average performance is measured when all jobs in the group are in stable state. Because of the performance protection (`LowPerf`), the performance loss caused by resource contentions among jobs are limited and even negligible in AUTOPS. Comparing to the performance from AUTOPS (in standalone mode), sharing AUTOPS among multiple jobs jobs may lose up to 9% training speed in our experiments. In some cases, the averaged performance of multiple jobs using AUTOPS is even better than the standalone performance from ps-lite.

To conclude, AUTOPS can reduce the number of Aggregators allocated for model aggregation through packing workload from multiple jobs. Meanwhile, it imports negligible performance loss to the training jobs.

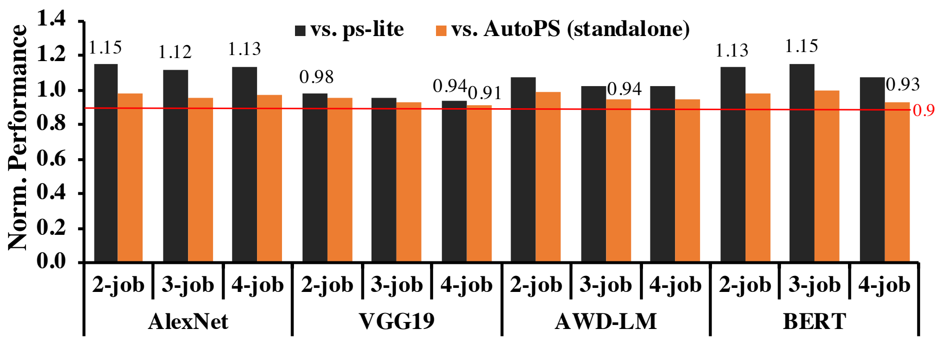


Figure 4.10: Performance impact when multiple (2s-2w) jobs share AUTOPS. Jobs in the same group train the same model. The averaged performance of each multi-job group is used. For comparison, it is normalized by the performance of the same job when using ps-lite and AUTOPS in standalone mode, respectively.

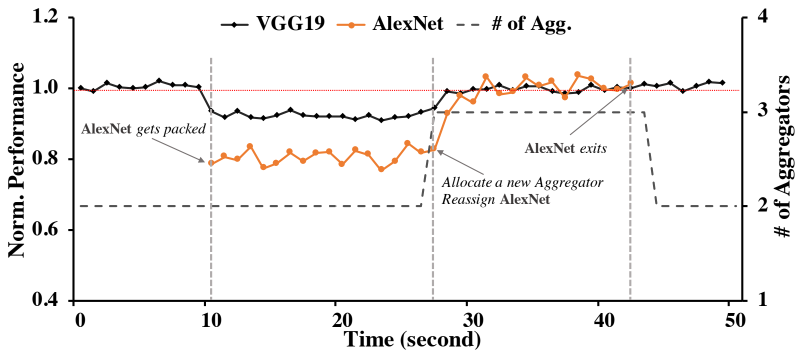


Figure 4.11: Time trace of the two-job case study. The training performance of the jobs is normalized by their standalone performance, respectively.

Case Study of Aggregator Scaling. We bring a case study with two jobs to show how AUTOPS scales Aggregator when job events (i.e., arrival and exit) occur. Figure 4.11 shows how the performance of jobs and Aggregator allocation is changed when job events occur. There is a VGG19 (2s-2w) job that uses AUTOPS for model aggregation and is already in steady state. Following its parameter server requirement, AUTOPS allocates 2 Aggregators for it. A new AlexNet (2s-2w) job just completed its initial performance profiling, and gets its first assignments to the two existing Aggregators at 11th second. The performance of VGG19 job is slightly affected because of resource contentions on those two servers. However, the new AlexNet job loses up to 22% of its performance. After monitoring enough iterations (default is 100), AUTOPS determines the assignments of AlexNet job should be revoked. At 27th second, AUTOPS allocates a new Aggregator and reassigns the AlexNet job. Both of the two jobs get better performance after that. The AlexNet job completes at the 42nd second. Since the newest Aggregator only has the model aggregations from AlexNet, AUTOPS releases it right after job exit.

4.5.2.4 Trace-Driven Simulation

We evaluate AUTOPS's performance in CPU saving using a real job trace. This is 10-week job trace from a 2000-GPU cluster in Microsoft. We compare the CPU consumption of AUTOPS against the CPU requirements of running jobs specified by users. Because ps-lite allocates the required amount of CPU resource for each job if it is used for model aggregation.

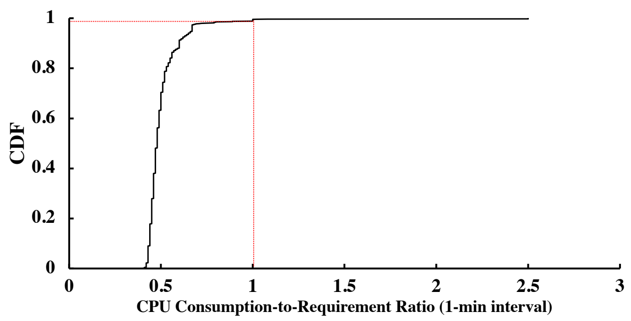


Figure 4.12: CPU consumption of AUTOPS compared to total CPU requirements of running jobs in the trace-driven simulation. The x-axis is the ratio of allocated CPU cores of AUTOPS to total CPU requirements of running jobs. CPU consumption and CPU requirements are measured with 1-min interval.

CPU Saving. We verify that the CPU saving benefits of AUTOPS still hold when it gets deployed in a large-scale cluster. Figure 4.12 compares the CPU consumption of AUTOPS against total CPU requirements of running jobs in the trace-driven simulation. The x-axis in the figure is the ratio of allocated CPU servers of AUTOPS to total CPU requirements of running jobs. Smaller value of this ratio means more CPU savings of AUTOPS. Over 99% of the time, this ratio is lower than 1, which means AUTOPS could save CPU resource for model aggregation for majority of the time. Very rarely, AUTOPS consumes more CPU resource than the CPU requirements from users. In worst cases, the ratio can be even larger than 2.5. These come from the scenarios that some allocated CPU resource in AUTOPS are idle because of recently-completed jobs. Due the periodical resource scaling in §4.3.3.4, AUTOPS could not release the free CPU resource until the end of current period, which makes CPU consumption of AUTOPS being higher than the CPU requirements of running jobs. Overall, AUTOPS could reduce the CPU cost (i.e., CPU time) of model aggregations by 52.7% in the simulated scenario

Revenue Benefit. From cloud provider's perspective, the saved CPU resource from AUTOPS can be rent for general purpose. Renting one CPU core in the cloud takes around $900/year [75]. Based on the CPU saving results from trace-driven simulation, the additional revenue contributed by AUTOPS in such a 2000-GPU cluster is around 1.5 million dollars for a 10-week period.

4.6 Discussion

Utilizing Transient Resource. The cloud clusters usually have some amount of transient resource that has short available time. Cloud providers often sell the transient resource to users as spot instances with discounted price [33]. With resource elasticity in Parameter Service, it is feasible to run model aggregations on spot instances for cost saving purpose. When the spot instance of a Aggregator is going to expire, other Aggregators could immediately take over the affect model aggregations through workload reassignment with negligible overhead.

Performance Isolation for Multitenancy. To pursue resource (CPU) efficiency, Aggregator may assign model aggregations from different jobs to the same Aggregator where jobs interfere with each other. Using a feedback-based assignment scheme, Parameter Service limits performance degradation caused by the interferences within a threshold (`LossLimit`). However, this mechanism is infeasible to the multi-tenant environment where jobs may have different requirements of performance. Parameter Service needs a performance isolation scheme which can satisfy the requirements from different users.

4.7 Related Work

Priority-based Model Aggregation ByteScheduler [141] is state-of-the-art data parallel training framework based on parameter server architecture. It optimizes the execution order of model aggregations according to the execution DAG of the training job and enforces their priorities in the data communication layer for less queuing delay. Parameter Service differs from ByteScheduler in two fundamental ways. First, the role of parameter server in ByteScheduler has been changed. The function of updating model parameters, which is originally executed by parameter server, is moved to the execution engine at worker side.

Therefore, the parameter server in ByteScheduler is just a hub that redistributes local gradients from workers. Second, the two PS-based systems working on different problems. Parameter Service focuses on reducing CPU consumption without sacrificing training performance. ByteScheduler [141] aims at improving training performance without considering CPU consumption. Moreover, comparing to ByteScheduler, Parameter Service has two benefits that are originated from the design of decoupling model aggregation. First, Parameter Service is easy to use since it is transparent to DL frameworks. However, users have to run the modified DL frameworks provided by ByteScheduler to receive performance improvement. Second, Parameter Service can utilize the transient resource in the cloud through its elastic feature. ByteScheduler is incapable of dynamically changing its underlying resource. Actually, the priority-based model aggregation scheme in ByteScheduler can be integrated into Parameter Service to improve training performance.

Alternatives for Model Aggregation. Horovod [152] applies bandwidth optimal ring-based AllReduce algorithms for model aggregation in DDL training. This approach can fully utilize the network bandwidth among the training nodes. But it requires homogeneous hardware (specially network links of equal bandwidth) which does not hold in the shared cluster. ParameterHub [119] is a physical machine designed for model aggregation. Similar to Parameter Service, ParameterHub provides a cluster-wise parameter hosting and aggregation function to multiple DDL training jobs. However, it is not cost-effective because it can not flexibly scale up or down according to the change of its load.

In-Network Model Aggregation With the trend of deploying programmable network devices in the cluster, in-network model aggregation has been proposed in the recent years. SwitchML [148] uses programmable switch dataplane to execute the model aggregation operations in DDL. It reduces the volume of exchanged data during model aggregation and network latency, which accelerate the training speed. iSwitch [114] is an in-switch aggregation acceleration solution for distributed reinforcement learning training. Focusing on the smaller but more frequent aggregations in reinforcement learning training, it conduces network packet level aggregation rather than the entire gradient vectors to reduce the aggregation overhead. SHARP [84] is a collective technology from Mellanox that is commonly applied for in-network aggregation. It's only available in certain InfiniBand switches and comes with fixed functions, which make it difficult to evolve to support new aggregation approaches.

Elastic Scaling on GPU Workers. Applying resource elasticity in GPU workers (i.e., dynamically adjusting training parallelism) could significantly improve GPU efficiency and shorter job completion time. However, existing DL frameworks either apply a fixed number of GPU workers throughout the lifetime of jobs, or adjusts the number of workers with high overheads that counteracts the benefits from elasticity. Recent work [135, 163, 98] has been proposed to cut down the overhead of scaling GPU workers. Furthermore, Or et al. [135] has an autoscaling engine, which considers account cost, other than GPU efficiency and job performance. Wu et al. [163] develops an elasticity-aware DL scheduler that achieves various scheduling objectives in multi-tenant GPU clusters. Hwang et al. [98] proposes an elastic resource sharing scheme for DL clusters. Its customized DL framework can transparently adjust the parallelization of jobs (i.e., number of GPU workers).

4.8 Conclusion

Parameter Service is an elastic model aggregation framework for emerging DDL training jobs. It could enhance the utilization of CPU, reserved for model aggregations, without hurting job performance. It exposes a shared model aggregation service. With that, multiple jobs can submit their model aggregations to the unique interface without allocating their own parameter servers. Internally, Parameter Service balances the tradeoff between resource efficiency and job performance through two knobs: dynamic workload and elastic resource scaling. Our implementation of Parameter Service, called AUTOPS, saves up to 75% of CPU servers when serving DDL training jobs. Its CPU saving benefits hold when it gets deployed in a large-scale cluster. More importantly, Parameter Service is totally transparent to user and can easily be adopted by popular DL frameworks.

CHAPTER V

Conclusions

5.1 Contributions

This thesis answers the question: *How to efficiently utilize the resources in a DL cluster and deliver good performance to the DL jobs on it?* We identify that the mismatches between DL jobs and resource management techniques in DL clusters are the root causes of severe resource and performance issues. Existing DL clusters ignore the new features of DL jobs and still apply the resource management techniques designed for traditional big-data clusters. We proposed and implemented a suite of new techniques to eliminate these mismatches and demonstrate that they can significantly improve resource-efficiency and the performance of DL jobs. More specifically, this thesis makes the following contributions:

Decentralized memory disaggregation for improving cluster-wise memory utilization: We built INFINISWAP, a decentralized memory disaggregation system, to improve the memory utilization of DL clusters. It opportunistically harvests and transparently exposes the unused memory in data preparation jobs to remote machines that are running out of memory. For scalability, INFINISWAP manages remote memory in a decentralized manner by leveraging the power-of-many-choices algorithms. To deliver good performance to the jobs, it utilizes a low-latency RDMA network to provide fast remote memory access. INFINISWAP can greatly improve cluster memory utilization and bring performance benefits to the data preparation jobs. INFINISWAP is easy to deploy and requires no modifications to applications, OSes, or hardware. To the best of our knowledge, INFINISWAP is the first runnable open-sourced memory disaggregation system and has inspired many follow-ups in the related research areas [153, 42, 44, 122, 109].

Two-dimensional attained-service-based scheduler for reducing the average JCT without complete job information: One important feature, which is often ignored, of DL training jobs, is that their execution time is mostly unpredictable. Without such information, the job schedulers in existing DL clusters are unable to minimize the average JCT of those jobs. We presented TIRESIAS, a GPU cluster manager tailored for DL training jobs. Its two-dimensional attained-service-based scheduler can efficiently schedule DL training jobs with partial or no prior knowledge of job execution time. In addition, the scheduler shows that considering both temporal and spatial information of DL training jobs is necessary for reducing their JCTs. TIRESIAS is the first GPU cluster manager that is capable of reducing the average JCT of DL training jobs without complete job information. It has shown significant improvements, in terms of the average JCT, over existing solutions.

Elastic model aggregation service for saving CPU cost: Existing DL clusters statically allocate CPUs to model aggregation in DDL training jobs. However, a considerable fraction of those CPUs is wasted due to the bursty nature of model aggregation. After characterizing the execution patterns of DDL training, we found the function of model aggregation could be decoupled from training jobs and be executed by third-party processes. Therefore, we implemented AUTOPS, an elastic model aggregation service, to reduce the CPU cost of model aggregation in a DL cluster. In AUTOPS, model aggregations from different jobs are efficiently packed to fit into the same group of CPUs. Also, AUTOPS can elastically scale up/down its CPUs based on the load on it. AUTOPS can significantly reduce the CPU consumption of model aggregation in DL clusters without sacrificing the training performance. The design of AUTOPS is transparent to users and compatible with popular DL frameworks.

5.2 Limitations

Even though the effectiveness of our proposed approaches has been shown in this thesis, the following limitations should be resolved or considered before applying them in real DL clusters.

INFINISWAP. Although INFINISWAP significantly outperforms the traditional disk-based swap space, its remote memory access latency is still much higher than the latency of local memory access, which can not bring performance transparency to many applications. A significant overhead was incurred because accessing a remote page still relies on the original data path in the kernel that was designed for low-speed disk-based swap space. For example, remote access requests have to go through the block I/O layer before being forwarded to INFINISWAP block device module. An efficient and lightweight data path from the kernel's memory controller to INFINISWAP is needed. Indeed, Leap [122], a direct follow-up to INFINISWAP has recently addressed this problem. The local backup disk in INFINISWAP block device is designed to handle remote failures (e.g., machine crash, network disconnection). If that happens, INFINISWAP will be as bad as the disk-based swap space. More advanced fault-tolerance mechanisms should be imported to protect data in remote memory from various failures without sacrificing the performance of INFINISWAP too much. Hydra [109], another follow-up to INFINISWAP brings the idea of *erasure-coded remote memory* to preserve fast remote memory access when remote failures occur.

TIRESIAS. The resource management scheme (2DAS) in TIRESIAS assumes homogeneous GPU devices in a DL cluster, which rarely holds in real scenarios. A DL cluster is often equipped with GPU devices with different computation and memory capabilities. One training job may have very different training performance (i.e., time to accuracy) on different GPU devices. Moreover, people may choose to use CPU to train some non-critical DL models, who do not care about their completion time, for cost-efficiency. Therefore, the proposed scheme should be extended to manage hybrid and heterogeneous computing devices for DL training. AlloX [108] is one of the most recent schemes that can manage both CPU and GPU resource for DL training. The primary design objective of TIRESIAS is to minimize the average JCT for a private DL cluster. It ignores the fairness among jobs and users, which is one of the most important service objectives in multi-tenant public clusters. Without fairness, some jobs or users may starve for a very long time even though the average JCT gets optimized by TIRESIAS. Recent solutions, such as Themis [120] and $Gandiva_{fair}$ [54], focus on how to fairly share the DL cluster among multiple jobs.

AutoPS. To migrate tensors with negligible overhead, AutoPS requires synchronized data-parallel distributed training. Without the synchronization constraint, workers may push their local gradients and pull back the updated model parameters at any time. Therefore, it will be very hard to find a time window for handling the overhead of tensor migration. In addition, AutoPS does not account for the network activities of model aggregation. In a DL cluster, the background network traffic can significantly slow down the data transmission in model aggregation, which indirectly wastes GPU and CPU resource. The cyclic execution design in AutoPS relies on the periodic iteration features of DL training. If the iteration duration of a job becomes longer and unpredictable due to network interference, the fixed execution cycle of the affected Aggregators will be jeopardized. Other jobs, which have model aggregations on the affected Aggregators, may suffer from performance degradation. Thus, future AutoPS needs to handle network interference.

5.3 Future Work

Following the success of DL technology, an increasing number of model serving applications with well-trained models are deployed in many domains. As a future research direction, one may focus on optimizing model serving systems running in DL clusters.

The objective of training a DL model is to learn the patterns from observed data that can be generalized to the unseen data. Model serving uses the pre-trained DL model to make predictions on new inputs (e.g., videos). Unlike model training which may take days or even weeks to complete, model serving is latency-sensitive and mostly used in user-interactive applications [63, 154]. The Service Level Objective (SLO) of a model serving system is to render a certain degree of prediction within the latency budgets specified by users. In addition, model serving systems have to efficiently leverage the hardware resource for reducing their operating cost as much as possible.

It is non-trivial for model serving systems to satisfy these two objectives together [147, 63]. Model serving systems often operate queries in batch for better resource efficiency, but a larger bath size means a longer execution time. DL models can be deployed on different hardware processors (e.g., CPU, GPU) which have very different processing capabilities and costs. Some serving systems choose to use inexpensive hardware (i.e., CPU), instead of the expensive GPU devices, as long as their latency requirements are satisfied [172, 147]. Moreover, each model can be assigned to multiple processors to meet the throughput requirements. Therefore, a serving system needs to be built by making trade-offs among latency, throughput, and cost (i.e., resource consumption) through configuration of its batch size, type, and amount of hardware resource.

GPU Multiplexing for Model Serving. Originally, one GPU device can only run a single DL model serving instance due to the limitations in GPU sharing. Therefore, it is hard to precisely allocate GPU resource for model serving applications. A fraction of resource may be wasted when a lightweight model serving instance is assigned to a powerful GPU device [167]. Multi-Process Service (MPS) [28] is a software feature designed for NVIDIA GPUs. It can statically partition the GPU resource among multiple serving instances. Recently, the new generation GPUs from NVIDIA come with a new hardware feature – Multi-Instance GPU (MIG) [27]. This technology allows multiple model serving instances to be physically isolated on the same GPU, greatly increasing the flexibility in

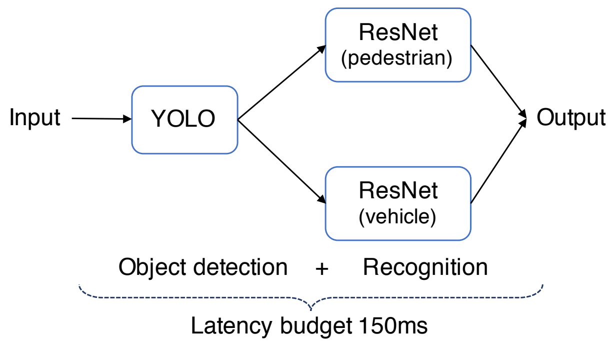


Figure 5.1: Prediction pipeline for traffic analysis application.

resource provisioning for model serving workloads.

However, it is not straightforward to adopt this new feature in existing model serving solutions [147, 63, 154, 64] that still manage GPU resource at device granularity. To fully leverage the benefits of fine-grained GPU slicing, the resource consumption of a model serving instance, which changes with the query load, has to be precisely predicted or monitored. A model serving system often has an elastic resource scaling feature to handle the variations in its query load. With MIG, the resource manager needs to choose between GPU device(s) and GPU slice(s). Although using GPU slices can reduce resource wastage when there is not sufficient query load, the model serving instances running on GPU slices render less resource-efficiency because of the much smaller batch sizes. On the other hand, the fragmented free GPU slices spread in the DL cluster could hurt resource utilization if the new instances need entire GPU devices instead of GPU slices.

Model Serving Pipeline with End-to-End Latency Targets. Other than involving a single DL model, prediction pipelines spanning multiple models are receiving increasing attention. A pipeline serving system must render a single complex query within its end-to-end latency budgets (SLO) and have enough capacity to meet the throughput requirements [154]. For example, a traffic analysis application needs to detect pedestrians and vehicles from traffic video. Figure 5.1 shows its prediction pipeline that has two stages: object detection and target recognition. The first one detects objects on the input image; the second one identifies whether the tagged objects are the targets (e.g., pedestrians, vehicles) or not. Note that two recognition models work in parallel in the second stage since the application requires two kinds of target objects. The pipeline serving system has to complete the two steps and gets the prediction result within $150ms$.

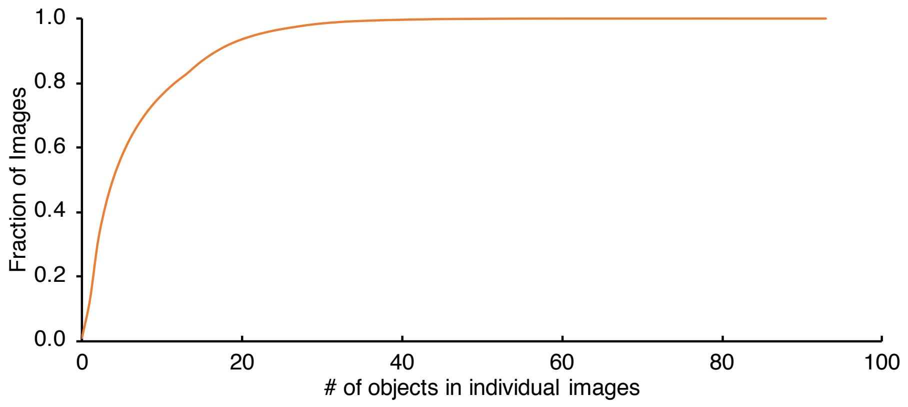


Figure 5.2: Number of objects detected in individual images in COCO dataset.

The complexity of configuring model serving systems grows when serving prediction pipelines. The end-to-end latency budget has to be split and assigned to the multiple pipeline stages so that each stage (i.e., model) can have its own latency budget. Latency splitting itself is a new configurable entity for prediction pipelines. The optimal configurations of individual models do not necessarily achieve the global optimization of the entire pipeline. Although the throughput target of the entire pipeline is specified in advance, downstream models do not yet have clear throughput objectives. In a prediction pipeline, the input of the downstream model is the intermediate results of its upstream models. In many cases, the number of results of a model is not fixed given different input data. For example, the input size/number of the vehicle recognition model in Figure 5.1 is determined by the number of detected objects in the initial traffic image. We measure the number of objects in individual images from COCO [118] dataset – a classic dataset for training object detection models – and find this number varying from 0 to more than 90 (Figure 5.2). On average, each image contains 7 objects. Among all the training images, 63% of them contain 6 or fewer objects. 33% of those images have 8 or more objects. Therefore, the intermediate load (i.e., input arrival rate of downstream models) of a prediction pipeline is hard to predict. Existing solutions such as Nexus [154] estimate the intermediate load using the averaged number of results of upstream models in the last cycle. However, this approach cannot approximate the intermediate load in real time. Most of time, it either overestimates or underestimates the intermediate load. The former leads to missing latency deadlines, while the latter renders resource underutilized. To the best of our knowledge, InferLine [64] is the only model serving system that considers this problem. It uses network calculus to make the scaling decisions for models in response to changes in their input arrival rate.

5.4 Summary

In this thesis, we first observe existing DL clusters are still applying the resource management techniques for traditional big-data clusters and ignoring the new features of DL jobs. Thus, we present INFINISWAP, TIRESIAS, and AUTOPS systems that can improve cluster memory utilization, reduce the average JCT of DL jobs, and save CPU cost of model aggregation, respectively. These systems sit in the different software layers of DL clusters and are transparent to DL jobs. Our proposed approaches are shown to mitigate the mismatches between DL jobs and resource management techniques in existing DL clusters and make notable improvements in both resource efficiency and job performance. These approaches and their findings have already influenced the related research area. Hopefully, this thesis will be useful for future development of resource-efficient DL clusters.

BIBLIOGRAPHY

[1] Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing. Accessed: 2017-02-02.

[2] AutoML. http://www.ml4aad.org/automl/.

[3] Amazon EC2 Elastic GPUs. https://aws.amazon.com/ec2/elastic-gpus/.

[4] GPU-Accelerated Microsoft Azure. https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/microsoft-azure/.

[5] Linux Multi-Queue Block IO Queueing Mechanism (blk-mq). https://www.thomas-krenn.com/en/wiki/Linux_Multi-Queue_Block_IO_Queueing_Mechanism_(blk-mq).

[6] Caffe. http://caffe.berkeleyvision.org/.

[7] CloudLab. https://www.cloudlab.us, .

[8] Graph Analytics Benchmark in CloudSuite. http://parsa.epfl.ch/cloudsuite/graph.html, .

[9] Fio - Flexible I/O Tester. https://github.com/axboe/fio.

[10] GPU on Google Cloud. https://cloud.google.com/gpu/.

[11] Google Container Engine. https://cloud.google.com/container-engine/, .

[12] Google Compute Engine Pricing. https://cloud.google.com/compute/pricing, . Accessed: 2017-02-02.

[13] YARN Capacity Scheduler. http://goo.gl/cqwcp5, .

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

[15] HP: The Machine. http://www.labs.hpe.com/research/themachine/, .

[16] HP Moonshot System: The world's first software-defined servers. http://h10032.www1.hp.com/ctg/Manual/c03728406.pdf, .

[17] Mellanox ConnectX-3 User Manual. http://www.mellanox.com/related-docs/user_manuals/ConnectX-3_VPI_Single_and_Dual_QSFP_Port_Adapter_Card_User_Manual.pdf, .

[18] Mellanox SX6036G Specifications. http://www.mellanox.com/related-docs/prod_gateway_systems/PB_SX6036G.pdf, .

[19] Intel RSA. http://www.intel.com/content/www/us/en/architecture-and-technology/rsa-demo-x264.html.

[20] Keras: Deep Learning for humans. https://github.com/keras-team/keras.

[21] Kubernetes. http://kubernetes.io.

[22] Linux memory management. http://www.tldp.org/LDP/tlk/mm/memory.html.

[23] Memcached - A distributed memory object caching system. http://memcached.org.

[24] Microsoft Azure Cloud Services Pricing. https://azure.microsoft.com/en-us/pricing/details/cloud-services/. Accessed: 2017-02-02.

[25] Apache MXNet. https://github.com/apache/incubator-mxnet.

[26] Accelio based network block device. https://github.com/accelio/NBDX.

[27] NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, .

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

[29] Protocol Buffers: Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. https://github.com/protocolbuffers/protobuf.

[30] ps-lite: A Lightweight Parameter Server Interface. https://github.com/dmlc/ps-lite.

[31] Pytorch: Tensors and Dynamic neural networks in Python. https://github.com/pytorch/pytorch.

[32] Apache Spark Data Validation. https://databricks.com/session/apache-spark-data-validation.

[33] Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot-instances/.

[34] Stackbd: Stacking a block device. `https://github.com/OrenKishon/stackbd`.

[35] Tensorflow: An End-to-End Open Source Machine Learning Platform. `https://github.com/tensorflow/tensorflow`.

[36] TensorFlow Benchmark Code. `https://github.com/tensorflow/benchmarks`.

[37] TPC Benchmark C (TPC-C). `http://www.tpc.org/tpcc/`.

[38] A Twitter Analog to PageRank. `http://thenoisychannel.com/2009/01/13/a-twitter-analog-to-pagerank`.

[39] VoltDB. `https://github.com/VoltDB/voltdb`.

[40] Samuli Aalto, Urtzi Ayesta, and Rhonda Righter. On the gittins index in the m/g/1 queue. *Queueing Systems*, 63(1-4):437, 2009.

[41] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

[42] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *USENIX ATC*, 2018.

[43] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-aware gpu scheduling for learning workloads in cloud environments. In *SC*, 2017.

[44] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[45] 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.

[46] G. Ananthanarayanan, A. Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, 2013.

[47] Eric A. Anderson and Jeanna M. Neefe. An exploration of network RAM. Technical Report UCB/CSD-98-1000, EECS Department, University of California, Berkeley, Dec 1994.

[48] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Scheduling: The multi-level feedback queue. In *Operating Systems: Three Easy Pieces*. 2014.

[49] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[50] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *NSDI*, 2015.

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

[52] John B Carter, John K Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *SOSP*, 1991.

[53] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[54] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *EuroSys*, 2020.

[55] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[56] Haogang Chen, Yingwei Luo, Xiaolin Wang, Binbin Zhang, Yifeng Sun, and Zhenlin Wang. A transparent remote paging model for virtual machines. In *International Workshop on Virtualization Technology*, 2008.

[57] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *SIGCOMM*, 2018.

[58] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[59] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[60] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM*, 2013.

[61] Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. An experimental time-sharing system. In *Spring Joint Computer Conference*, pages 335–344, 1962.

[62] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2C2: A network stack for rack-scale computers. In *SIGCOMM*, 2015.

[63] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.

[64] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491, 2020.

[65] David E Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten Von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing*, 1993.

[66] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P Xing. High-performance distributed ml at scale through parameter server consistency models. In *AAAI*, 2015.

[67] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[68] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[69] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[70] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *EuroSys*, 2016.

[71] Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *IPPS/SPDP*, 1999.

[72] Michael J Feeley, William E Morgan, EP Pighin, Anna R Karlin, Henry M Levy, and Chandramohan A Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, 1995.

[73] Michael J Feeley, William E Morgan, EP Pighin, Anna R Karlin, Henry M Levy, and Chandramohan A Thekkath. Implementing global memory management in a workstation cluster. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 201–212. ACM, 1995.

[74] Edward W. Felten and John Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, University of Washington, Mar 1991.

[75] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *NSDI*, 2018.

[76] Michail D Flouris and Evangelos P Markatos. The network RamDisk: Using remote memory on heterogeneous NOWs. *Journal of Cluster Computing*, 2(4):281–293, 1999.

[77] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[78] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[79] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, 2013.

[80] John Gittins, Kevin Glazebrook, and Richard Weber. *Multi-armed bandit allocation indices*. John Wiley & Sons, 2011.

[81] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert Nicholas Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *OSDI*, 2016.

[82] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[83] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[84] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*.

[85] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[86] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[87] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[88] Hayit Greenspan, Bram Van Ginneken, and Ronald M Summers. Guest editorial deep learning in medical imaging: Overview and future promise of an exciting new technique. *IEEE Transactions on Medical Imaging*, 35(5):1153–1159, 2016.

[89] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *NSDI*, 2019.

[90] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *SIGCOMM*, 2016.

[91] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, 2013.

[92] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Addressing the straggler problem for iterative convergent parallel ml. In *SoCC*, 2016.

[93] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE CVPR*, 2016.

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

[95] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 2012.

[96] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[97] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, 2015.

[98] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic resource sharing for distributed deep learning. In *NSDI*, 2021.

[99] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. *arXiv preprint arXiv:1901.05758*, 2019.

[100] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *USENIX ATC*, 2019.

[101] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[102] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, 2016.

[103] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, 2014.

[104] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[105] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[106] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[107] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, 2019.

[108] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: compute allocation in hybrid clusters. In *EuroSys*, 2020.

[109] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G Shin. Mitigating the performance-efficiency tradeoff in resilient memory disaggregation. *arXiv preprint arXiv:1910.09727*, 2019.

[110] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. In *SIGMOD*, 2016.

[111] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, 7(4):321–359, 1989.

[112] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[113] Tian Li, Jie Zhong, Ji Liu, Wentao Wu, and Ce Zhang. Ease. ml: Towards multi-tenant resource sharing for machine learning workloads. *Proceedings of the VLDB Endowment*, 11(5):607–620, 2018.

[114] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating distributed reinforcement learning with in-switch computing. In *ISCA*, 2019.

[115] Shuang Liang, Ranjit Noronha, and Dhabaleswar K Panda. Swapping to remote memory over Infiniband: An approach using a high performance network block device. In *Cluster Computing*, 2005.

[116] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, 2009.

[117] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. System-level implications of disaggregated memory. In *HPCA*, 2012.

[118] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[119] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. *arXiv preprint arXiv:1805.07891*, 2018.

[120] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *NSDI*, 2020.

[121] Evangelos P Markatos and George Dramitinos. Implementation of a reliable remote memory pager. In *USENIX ATC*, 1996.

[122] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *USENIX ATC*, 2020.

[123] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[124] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

[125] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. Analysis of the memory registration process in the Mellanox Infiniband software stack. In *Euro-Par*, 2006.

[126] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. *arXiv preprint arXiv:1706.04972*, 2017.

[127] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX ATC*, 2013.

[128] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[129] Michael Mitzenmacher, Andrea W. Richa, and Ramesh Sitaraman. The power of two random choices: A survey of techniques and results. *Handbook of Randomized Computing*, pages 255–312, 2001.

[130] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[131] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, 2015.

[132] Tia Newhall, Sean Finney, Kuzman Ganchev, and Michael Spiegel. Nswap: A network swapping module for Linux clusters. In *Euro-Par*, 2003.

[133] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[134] Misja Nuyens and Adam Wierman. The Foreground–Background queue: A survey. *Performance Evaluation*, 65(3):286–307, 2008.

[135] Andrew Or, Haoyu Zhang, and Michael Freedman. Resource elasticity in distributed deep learning. In *MLSys*. 2020.

[136] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high performance storage entirely in DRAM. *SIGOPS OSR*, 43(4), 2010.

[137] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[138] Gahyun Park. Brief announcement: A generalization of multiple choice balls-into-bins. In *PODC*, 2011.

[139] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[140] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*, 2018.

[141] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *SOSP*, 2019.

[142] Russel Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[143] Pramod Subba Rao and George Porter. Is memory disaggregation feasible?: A case study with Spark SQL. In *ANCS*, 2016.

[144] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid training data creation with weak supervision. *PVLDB*, 2017.

[145] Charles Reiss. *Understanding Memory Configurations for In-Memory Analytics*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2016.

[146] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[147] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. Infaas: A model-less inference serving system. *arXiv preprint arXiv:1905.13348*, 2019.

[148] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *NSDI*, 2021.

[149] Frank B Schmuck and Roger L Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, 2002.

[150] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[151] Ziv Scully, Mor Harchol-Balter, and Alan Scheller-Wolf. Soap: One clean analysis of all age-based scheduling policies. In *SIGMETRICS*, 2014.

[152] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[153] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, 2018.

[154] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[155] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[156] Radu Stoica and Anastasia Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. In *Ninth International Workshop on Data Management on New Hardware*, 2013.

[157] Peng Sun, Yonggang Wen, Nguyen Binh Duong Ta, and Shengen Yan. Towards distributed machine learning in shared clusters: A dynamically-partitioned approach. In *IEEE Smart Computing (SMARTCOMP)*, 2017.

[158] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*, 2016.

[159] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC*, 2013.

[160] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. The power of choice in data-aware cluster scheduling. In *OSDI*, 2014.

[161] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with Borg. In *EuroSys*, 2015.

[162] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[163] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, and James Cheng. Elastic deep learning in multi-tenant gpu cluster. *arXiv preprint arXiv:1909.11985*, 2019.

[164] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI*, 2018.

[165] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *OSDI*, 2020.

[166] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. An introduction to computational networks and the computational network toolkit. Technical report, Microsoft Research, October 2014.

[167] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained gpu sharing among cnn applications. 2020.

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

[169] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

[170] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *USENIX ATC*, 2017.

[171] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *ACM SoCC*, 2017.

[172] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *USENIX ATC*, 2018.

[173] Qi Zhang, Mohamed Faten Zhani, Shuo Zhang, Quanyan Zhu, Raouf Boutaba, and Joseph L Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *ICAC*, 2012.

[174] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hocarbyneng Tang, and Jie Xu. Fuxi: A fault-tolerant resource management and job scheduling system at internet scale. In *VLDB*, 2014.

[175] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming

Zhang. Congestion control for large-scale RDMA deployments. In *SIGCOMM*, 2015.

[176] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *ICCV*, 2015.