CLEI Electronic Journal, Volume 28, Issue 6, paper 2, October 2025

Optimizing Resource Allocation for Elastic Tensorflow Container Scheduling on Multi-core Servers

Leandro Libutti
Instituto de Investigación en Informática LIDI (III LIDI)
Facultad de Informática, UNLP CIC
La Plata, Argentina, 1900
llibutti@lidi.info.unlp.edu.ar

and

Marcelo Naiouf
Instituto de Investigación en Informática LIDI (III LIDI)
Facultad de Informática, UNLP CIC
La Plata, Argentina, 1900
mnaiouf@lidi.info.unlp.edu.ar

and

Francisco D. Igual
Departamento de Arquitectura de Computadores y Automática
Universidad Complutense de Madrid
Madrid, España, 28001
figual@ucm.es

Abstract

In this paper, we delve into the necessity of applying coupled application-container elasticity in the context of resource allocation within multi-tenant scenarios, specifically targeting CNN training procedures on multi-core architectures. Our focus is primarily on elucidating the process of endowing a parallel application, exemplified by Tensorflow, with dynamic thread elasticity support. Additionally, we emphasize the criticality of harnessing this functionality in tandem with container resource management by container orchestrators to avert undesirable resource oversubscription and under-utilization effects, all the while optimizing core occupancy. The empirical findings comprise a wide range of resource allocation and dynamic re-allocation strategies, and reinforce the compelling case for elasticity support not only at the container level but also at the application level. These findings underscore the substantial improvements achieved in terms of time-to-solution, resource utilization, and productivity across diverse workloads.

Keywords: Multi-Core Server, Orchestrators, Elastic Tensorflow, Container, Docker

1 Introduction

As modern multi-core and many-core servers continue to evolve with an increasing number of general-purpose cores, effectively utilizing these resources from a single application becomes a significant challenge. In order to address this challenge, containerized applications are often co-scheduled for execution, following a strategy known as *horizontal scalability* [1], increasing resource usage by just replicating the execution of new applications. At the same time, confining applications in containers offers a plethora of mechanisms to isolate and assign resources (e.g. processing cores) to applications in an easy and safe manner [2, 3].

In multi-tenant deployments, containerized applications can also be scaled *vertically* [4, 5]. This involves

statically defining the number of cores (or other compute resources) assigned to a specific container upon creation and configuring the application within the container to optimally utilize these local resources. However, vertical scalability is typically a *static* process, with resource assignment remaining unchanged throughout the container's lifetime.

The rigidity in the assignment of resources to applications/containers is not itself a problematic feature [6, 7]. Static resource assignment offers benefits such as isolation, resource contention control, and determinism, but it becomes problematic in environments with multiple heterogeneous applications running concurrently. In such environments, resources are dynamically released or requested, and the lack of elasticity in applications leads to situations of oversubscription or under-utilization of resources [7].

This paper explores the need for *coupled* mechanisms to dynamically manage both container resource allocation and the applications within them. To illustrate this need, we use a specific use case involving the deployment of multiple TensorFlow training containerized applications for neural networks, scheduled concurrently on a modern multi-core server.

Our main contributions are as follows:

- We provide evidence demonstrating that oversubscription or under-utilization occurs when container resources are modified without corresponding adaptations in the containerized applications. Our illustrative example is presented in Section 3.
- We describe necessary changes to a specific application (Tensorflow) to enable on-demand resource elasticity for a particular resource type (number of CPU cores used by the application). These techniques are detailed in Section 4.
- We propose a scheduling engine equipped with elasticity support. This engine schedules containerized applications for execution and supports dynamic resource assignment or reassignment at both the initial execution point and during the container's lifetime. The scheduler can modify both the usage of containers and applications resources. Section 5 outlines the structure of schedulers and various scheduling and resource assignment policies.
- We present evidence of the productivity, time-to-completion, and resource usage advantages of coupled container-application resource management compared to static resource allocation in modern multi-core architectures under different workload profiles (Section 6).

2 Background and related work

2.1 Deep Learning Frameworks

Deep Learning frameworks provide a user-friendly interface for creating, training, and testing models in diverse application domains such as image and speech recognition or natural language processing. These frameworks are usually able to exploit different compute resources, such as multi-core CPUs, GPUs, TPUs (Tensor Processing Units), and NPUs (Neural Processing Units) under user requests. They encapsulate mathematical operations in high-level libraries that provide abstraction to the developers and allow to adjust basic applications according to customer needs. Also, they allow to use datasets in a simple way, just by adding the necessary preprocessing depending on the specific target model. In order to improve the performance of the model execution (in both the training and the inference stages), these frameworks usually include parameters to define the amount of resources under usage on demand (e.g. determining the degree of parallelism and hence the use of compute cores or number of GPUs when processing the model).

Convolutional neural networks (CNNs) are nowadays one of the main families of deep learning models targeting a variety of applications; CNNs are determined by the characteristics of their layers. For CNNs, convolution layers typically take up the most of the inference time, with the remaining time spent on batch normalization (BN), activation, fully connected (FC), and pooling layers. In a convolution layer, most of the data (activations and filter weights) are reused with a sliding window, so it has a very high compute-to-memory bandwidth ratio (i.e., they are highly compute intensive) [8]. In both convolution layers and FC layers, the available parallelism is usually massive, and hence the final performance obtained in a multi-core server is ultimately determined by the amount of execution units in use by the model. This resource assignment ultimately determines the job completion time (JCT) of these models and needs to be carefully tackled in massively parallel architectures.

2.2 Model containerization

Containerization is a lightweight virtualization method that enables applications to provide services from a sandbox runtime environment without launching the overhead of a virtual machine [9]. The most common

framework is Docker [10]. Docker bundles everything an application needs to run, including code, libraries, runtime, framework, and devices to easily build and test applications [11]. The new containers can be initialized on a physical node by sending commands to the local docker daemon. The dependencies or prerequisites of DL models can be packaged into a Docker image (called Dockerfile).

Docker provides a set of commands to control the execution of DL containers, including launching a new container with a set of pre-assigned resources (e.g. number of cores and amount of memory), pausing and monitoring resource usage, or updating dynamically the amount of resources assigned to the container. Replicating container execution favors *horizontal scalability*; dynamically or statically managing the resources assigned to a container favors and enables *vertical scalability*.

2.3 Container schedulers and orchestrators

To enable dynamic monitoring and control of multiple containerized applications, an orchestrator/scheduler is required to perform this task. Container orchestration deals with runtime management to support deployment, execution, and maintenance. It typically provides resource limit control, load balancing, state check, fault tolerance, and automatic scaling [2].

These type of orchestrators and their integrated schedulers are mainly composed of four components to manage different container features:

- The *resource controller* allows defining the amount of CPU and memory allocated to each container, considering the limits imposed in the configuration to avoid interference between all active containers.
- The *scheduler* defines the number of containers to be executed per node. Affinity policies can be defined for each node. This allows different workloads to be defined to achieve a distribution that increases the overall performance of the system.
- The *load balancer* distributes the workload among multiple container instances. There are different balancing policies. The most commonly used is round-robin. New load balancing policies can be included if the orchestrator allows it.
- The *life analyzer* checks port connections (TCP/UDP/SSH) and checks that requests are received and sent over the network.

Several container orchestrators can be found with local or cloud solutions. Local solutions include Docker *swarm* [12], a native environment that includes container clustering and system administration functionalities through the Docker engine. In addition, Kubernetes [13] is an orchestration system for Docker containers (with support for other container and virtual machine technologies) that allows workloads to be scheduled and managed based on user-defined parameters. Marathon [14], a container orchestrator for Apache Mesos [15] with features that allow running applications in a clustered environment. In terms of cloud solutions, Cloudify [16], an orchestrator that allows the modeling of applications and services, and the automation of their life cycle.

2.4 Related Work

Researchers usually focus on resource allocation on GPUs, as they are currently the most widely used architecture for training DL models. However, it is important to consider memory and CPU consumption as it directly affects training performance.

CODA [17] shows the contention that occurs on the CPU if jobs are placed on the same compute node as DL jobs. Although most of the intensive computation is performed on the GPU, it is important to note that training jobs require the CPU for pre-processing and gradient synchronization. CODA shows that it is very important to properly adjust the number of cores used on the CPU to avoid contention with other jobs. To this end, they characterized several representative DL models as functions of the amount of GPU and CPU resources. From this, they developed a model to eliminate real-time contention showing a 20% improvement in GPU-CPU utilization.

Synergy [18] considers the impact of the amount of CPU and memory resources allocated to the DL models. It proposes a DNN scheduler that performs allocations using optimistic profiling. This technique analyzes the different memory and CPU configurations looking for the one that offers the best performance to the DL model.

In reference to the use of schedulers and scheduling policies to improve job completion times and resource utilization for DL models, there are several related research studies. Optimus [19] proposed a customized job scheduler for deep learning clusters, improving training times. It was implemented on top of Kubernetes, where it allows to dynamically allocate cluster resources. This scheduler was executed with the MXNet

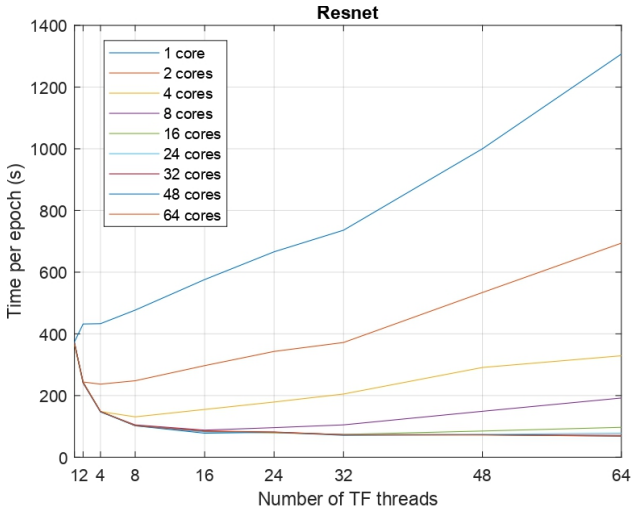


Figure 1: Motivational example: effects of under-utilization and oversuscription in Tensorflow.

framework. Gandiva [20] proposed a scheduling framework that improves latency and efficiency of DL deep learning models in GPU clusters. This work exploits intra-job predictability to split GPU time efficiently across multiple jobs, thus providing low latency. Through a prototype implementation and comparisons, Gandiva demonstrates that it can accelerate hyper-parameter search during deep learning by up to one-tenth. MARBLE [21] takes into account the non-linear scalability of GPUs in clusters at the intranode level. It avoids the low GPU utilization of multi-GPUs in HPC systems when GPU resources are shared among multiple DL training. Reduce job completion time compared with Platform Load Sharing Facility (LSF) and Optimus schedulers. Wang et al. [22] developed a non-intrusive scheduling framework combined with Kubernetes and Tensorflow. It includes elastic GPU allocation based on a "SideCar" process that temporarily stops and restarts the DL training jobs with different resources. SLAQ [23] runs distributed ML jobs in a cluster in the feedback phase maximizing the average model accuracy. Dynamically allocates CPU resources based on the demand of the ML jobs.

3 Oversubscription and under-utilization of resources

In this section, we present a scenario involving a multi-core server with 64 cores, where applications are confined within containers. Each container, called *task*, is allocated a specific number of CPU cores as execution resources. A centralized task scheduler manages scheduling and resource assignment, allowing potential resource reassignment at runtime, demonstrating container-level elasticity.

Our experiment aims to highlight the impact of resource overutilization and under-utilization within a container. We increase the number of CPU threads requested by the application confined in the container without modifying the number of cores allocated to the container. This allows us to illustrate the performance degradation resulting from oversubscription. Conversely, we also show the effect of increasing the number of cores assigned to a container without increasing the application's parallelism, leading to resource under-utilization.

Figure 1 provides visualizations of the results of our experiment, illustrating the effects of oversubscription and under-utilization resources in TensorFlow applications. The figure demonstrates how the execution time decreases until the requested CPU cores match the allocated resources, beyond which performance declines due to oversubscription. Additionally, the static nature of the application's parallelism prevents the full utilization of assigned cores, resulting in resource under-utilization.

The message extracted from this experiment is clear: in scenarios with containerized multi-thread applications, assigning resources (statically or dynamically) to containers is not enough if applications confined within them do not modify their behavior accordingly. Increasing the amount of resources assigned to a container will derive to those resources being infra-utilized; decreasing the amount of resources assigned to the container will drive to oversuscription. In all cases, a *coupled* resource assignment and exploitation strategy between the container and the application, managed by an external entity/orchestrator, becomes mandatory. This strategy, applied to Tensorflow, is explained next.

4 Elasticity in Tensorflow

4.1 Tensorflow execution model

Tensorflow represents Machine Learning models through computational graphs (specifically, directed acyclic graphs, or DAG's). DAG nodes describe mathematical operations and the edges represent the data (tensors) flowing between them. DAG's are common in all parallel library implementations, and are known to exhibit excellent scalability and core occupation capabilities when mapped to modern multi-core processors. Tensorflow integrates two main components in DAG execution:

- **Master thread**: divides the DAG among the worker threads. Splits the DAG and assigns the execution of each sub-DAG to the available (or requested) devices.
- **Worker threads (one per execution device)**: Worker threads (WT's) are bound to one or more execution devices (cores, groups of cores, or GPU's). WT's receive requests from the master thread, schedule the execution nodes of the sub-DAG, and monitor the associated device(s).

The master thread, after analyzing the initial graph, distributes the ready nodes to the available worker threads in a first scheduling stage.

Each worker thread is bound to two different queues. The Q_{ready} queue receives the newly released nodes released by a previous execution of a given node on the same thread. These tasks are locally analyzed and different criteria are applied to them before either *(i)* scheduling them for execution by the same worker thread, and hence adding them to a final local execution queue (Q_{inline} hereafter); or *(ii)* scheduling them to a different thread (and hence located in a remote Q_{inline} queue). In more detail, each thread repetitively applies the following procedure for scheduling ready nodes.

1. Checks whether $\mathrm{Q_{ready}}$ is empty. If not, proceed with STEP 2. Otherwise, the scheduling stage is completed.
2. The next node ($\mathrm{N_{next}}$) is extracted from $\mathrm{Q_{ready}}$.
3. If $\mathrm{N_{next}}$ is a *cheap* node (it does not require intensive computation), it is queued in the local Q_{inline} marking it as ready for local execution.
4. If the node is not expensive (in time), and the current worker thread already has another task of this type ready for execution, it is delegated to another thread. The scheduling algorithm returns to STEP 1.

The scheduling of nodes from Q_{inline} for execution proceeds as follows:

1. If Q_{inline} is not empty, its head is extracted, and the node proceeds with execution. Otherwise, the worker thread remains inactive waiting for ready tasks.
2. The next node in Q_{inline} is extracted and analyzed for execution.
3. The node is executed using the specific kernel implementation for the target device (e.g. CPU, GPU, or accelerator).
4. Once the execution has been completed, the released dependencies are notified to the master thread.
5. The thread checks whether there are new nodes to schedule. If this check is affirmative, the scheduling procedure for the ready tasks described above proceeds from STEP 1.

4.2 Parallelization parameters

TensorFlow's DAG-based execution model allows parallel execution of nodes. Hence, the framework integrates mechanisms to control the exploitation of parallelism in two levels:

- **Intra parallelism**. Defines the number of threads used for the execution of an individual node (e.g. a convolution operation, matrix multiplication, etc.), if the kernel is designed to be multi-threaded.
- **Inter parallelism**. Defines the maximum number of independent nodes that can be executed simultaneously.

Both types of parallelism can be selected by the user at runtime, but the selection is *static*, fixed upon the whole DAG creation, and constant throughout its execution. They are defined using specific functions in Python, such as *set_intra_op_parallelism* and *set_inter_op_parallelism*. In practice, Tensorflow delegates the execution of both levels of parallelism to two external actors when using CPU cores: the Eigen library [**?**] to manage a thread pool of workers in the case of inter-parallelism or intra-parallelism when no external libraries are used, or underlying mathematical libraries (e.g. Intel oneMKL) for intra-parallelism when they support threaded implementations.

In Eigen, a thread pool is created upon the invocation of Tensorflow and remains active throughout the complete execution, without any possibility of modifying the number of threads within it, or their individual activation/deactivation. Similarly, the number of threads defined for intra-parallelism and communicated to underlying libraries is also fixed.

4.3 Elasticity integration in Tensorflow

Armed with the previously described parallel execution model, the introduction of malleability/elasticity within the Tensorflow framework requires modification in two different components: the Eigen library (specifically the management of thread status in the thread-pool) and the execution model of Tensorflow.

4.3.1 Eigen library changes

The Eigen library has been modified to add a *status* flag in each thread from the thread pool; this flag allows blocking or waking up threads on demand.

The thread controller element in Eigen is able to admit modification requests in the degree of parallelism. Upon reception, two different cases can occur:

1. **Request for increase of parallelism**. In this case, the first check consists in determining if the request can be fulfilled by modifying the status of the current threads *idle* and *active*. If so, *idle* threads are activated to satisfy the requirements.

2. **Request for parallelism reduction**. The first check is to determine whether the rollback request will leave at least one worker thread active. If the answer is positive, the status of the requested number of threads changes to *idle*. If the queues of these threads have pending tasks, they will be served by the remaining *active* threads.

When a thread starts the waiting procedure for new ready nodes, it evaluates whether it needs to remain *active* or become *idle* depending on its current (possibly modified) status. In the latter case, it can proceed with blocking if Q_{ready} and Q_{inline} are empty. Otherwise, before blocking, the remaining tasks are delegated to an active worker thread.

4.3.2 Tensorflow core changes

Starting with *Tensorflow* 2.0 , the *thread* state cannot be controlled. The only possible communication delegates the execution nodes to another *thread*. Additional control is added in the TF core, specifically in the logic responsible for distributing the execution nodes among the worker threads explained in Section 4.1. The thread status query is added in the scheduling node stage. $\mathrm{Q_{ready}}$ nodes are scheduled in the worker thread if their state is active and the nodes are not expensive in time. Otherwise, the following cases are considered:

- The *worker thread* is active, and the nodes are expensive in time: we delegate the nodes to another *worker thread*, as the original scheduling does.

- The *thread* is not active: Delegate the node to another *worker thread* .

4.3.3 On-demand malleability control

To increase or decrease the intra and inter parallelism is necessary to add an external communication mechanism by which the user of the framework can modify it at any time during the execution of the algorithm. For this, we added POSIX signal handling in the *executor* component of each worker thread. In our implementation, two types of POSIX signals are used to request the increase or decrease of the amount of parallelism externally.

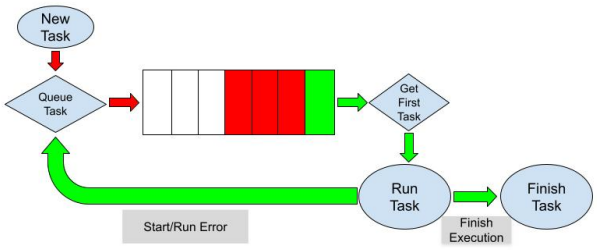


Figure 2: Scheduling process in FCFS policy

5 Scheduling elastic containers

Our elasticity-aware scheduling of applications (confined in containers) is based on a piece of software that manages the scheduling and resource assignment departing from a set of execution requests from the user. Thus, the scheduler departs from a scenario in which one or multiple users request the execution of applications with an associated amount of execution resources. At this stage, these resources are limited to execution cores.

The scheduling logic includes two main components:

1. The *scheduling server* receives execution requests from the user, including execution parameters (e.g. model, number of epochs, steps per epoch or training dataset for Tensorflow executions) and requested execution resources. The server schedules the request according to a specific *scheduling algorithm*, allocates/reallocates resources to any container, and attends to different *resource assignment policies*.

2. The *clients* are automatically deployed inside each container and kept alive while the container is running. Each client communicates with the scheduler server in a bidirectional way, receiving requests for resource modification, and notifying special situations (mainly errors in execution and starting/ending points in TF application). The client translates incoming resource modification requests into POSIX signals. Then, client sends notification to the TF application to modify the internal parallelism using the elasticity explained in Section 4.3.

The communication between server and clients is performed via sockets, and occurs under two situations: first, when a task finishes and resources are released, the scheduler client notifies server to indicate that those resources can be reassigned to existing or new tasks; second, the server notifies the corresponding client when reassignment resources is necessary. In some situations, for example, if a preemptive round-robin scheduling policy is applied, this communication can also be performed periodically.

5.1 Scheduling policies

Given a pool of application execution requests, the scheduling server places them in a queue or queues waiting for scheduling. The design of queues depends on the scheduling policy in place; currently, the server supports FCFS and Priority policies, with optional preemptive versions when possible.

5.1.1 FCFS policy

FCFS is the basic policy implemented in the scheduler. To manage this policy, we use a FIFO queue (first come, first served) that maintains the order of arrival of the requests. Fig. 2 shows the scheduling process using this policy. Each new task (or deep learning model) requested by a client is queued at the end. The scheduler server takes the oldest request in the queue and executes it. If an error occurs in the launch or execution of the deep learning model, the request returns to the end of the FIFO queue.

5.1.2 Priority policy

Priority handles a FIFO queue for each priority defined in the scheduler server. Fig. 3 shows an example of scheduling process. Two priorities are selected; two FIFO queues are created, one for each priority. The scheduler server always takes a new task from the highest priority queue that has a pending task to be launched. The handling of errors in the launch and execution of a task is in accordance with the FCFS policy. When the task fails, it is queued again at the end of the assigned priority queue.

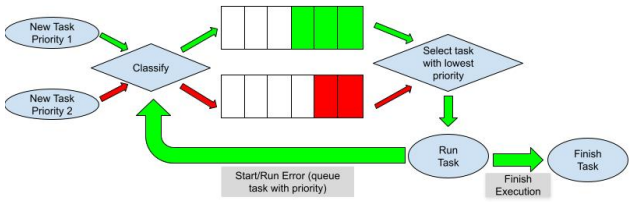


Figure 3: Scheduling process in Priority policy

5.2 Resource assignment and re-assignment policies

Independently from the selected scheduling policy, a second decision prior to the final mapping of a scheduling task to a specific core or group of cores consists in assigning the amount of available resources to it. This resource assignment and reassignment process is fixed at two different points: *(i)*, at the initial scheduling point, and *(ii)*, every time a task finishes its execution, releasing resources. The proposed policies include the following.

- STRICT: the amount of resources assigned to a container must always be equal to that originally requested for the task. In case there are not enough resources available at the scheduling point, the task is not scheduled and remains in the queue (typically, it is inserted again at the end of the waiting queue).
- ALWAYS_ATTEND: if the amount of resources requested for the task is greater than those available at the scheduling point, the task is still scheduled, assigning the available resources to it.
- MAX_PROP: If the amount of resources requested for the task is larger than those available at the scheduling point, the task is still scheduled, redistributing the overall resources currently used by running tasks to fulfill a proportional partition depending on the original set of resource requests.

5.3 Resource assignment and preemption

We leverage the mechanisms offered by Docker in order to manage both the resource assignment (initially assigned or reassigned at any execution point) and to emulate the preemption of a given container. For the first, we impose resource constraints at creation time (`docker run`)[1] or at any point of the container lifetime, updating its configuration (`docker update`[2]). The `pause`/`wake-up` mechanism in Docker[3] is the way in which the scheduler can release (temporarily) the resources assigned to a container, effectively emulating preemption.

6 Experimental results

6.1 Target architecture

The target server is based on a dual-socket Intel Xeon Platinum 8358 processor comprising 32 physical cores per socket (HyperThreading was disabled during our experiments), running at 1.7 GHz to avoid undesired frequency throttling effects. The server is equipped with 256 GB of DDR4 RAM memory. The operating system is based on a Linux kernel version 5.10. Docker version 20.10.17 was used in all experiments. The TensorFlow version is 2.0 without the *eager* DAG execution mode.

6.2 Workload definition and metrics

The workloads selected for the evaluation are based on training procedures for the Resnet-50, EfficientNet, and ConvNet models, with 5 training epochs and 20 steps per epoch. The selected data set is CIFAR-100, with input images of dimension 32×32 and three channels. The training batch size was fixed to 128.

Four different scheduling scenarios are used, modifying the usage of TF application resources and the number of simultaneous container requests. The specifications for the selected workloads are as follows:

[1] `https://docs.docker.com/config/containers/resource_constraints/`
[2] `https://docs.docker.com/engine/reference/commandline/update/`
[3] `https://docs.docker.com/engine/reference/commandline/pause/`

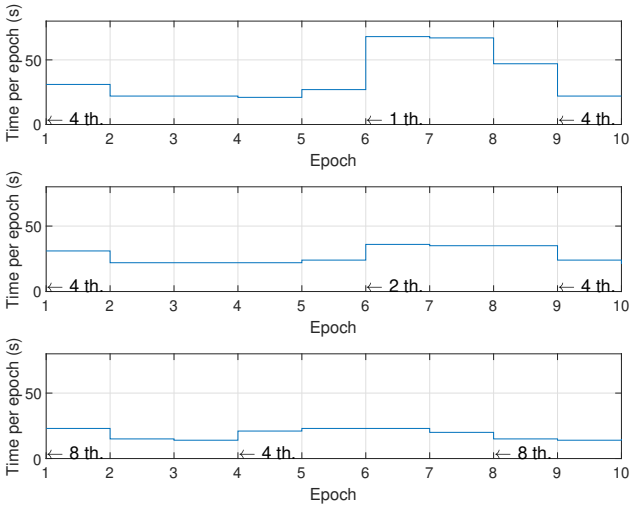


Figure 4: Example of intra-parallelism modification in Tensorflow for three different elasticity scenarios for a long-running Tensorflow training procedure.

- L_{RES}-H_{REQ}: Core requests per application are randomly selected in the range 1-4. The requested arrival time is 15 seconds for Resnet, 5 seconds for ConvNet, and 10 seconds for EfficientNet.
- H_{RES}-H_{REQ}: Core requests per application are randomly selected in a range of 5-8. The requested arrival time is 15 seconds for Resnet, 5 seconds for ConvNet, and 10 seconds for EfficientNet.
- H_{RES}-L_{REQ}: Core requests per application are randomly selected in a range of 5-8. The requested arrival time is 60 seconds for Resnet, 40 seconds for ConvNet, and 50 seconds for EfficientNet.
- L_{RES}-L_{REQ}: Core requests per application are randomly selected in the range 1-4. The requested arrival time is 60 seconds for Resnet, 40 seconds for ConvNet, and 50 seconds for EfficientNet.

Results are presented hereafter in terms of different metrics that evaluate the efficiency of each scheduling policy. The scheduler is able to gather metrics that allow reporting, on a per-task basis or averaged throughout an overall execution period. These metrics include TME (execution time of the task); TMR (time to first schedule of the task); TMER (sum of the two previous metrics); RA (ratio between the amount of resources used by the containers and the total presented by the system); and PROD (productivity, number of containers completed per time unit).

6.3 Evaluation of elasticity in Tensorflow

To demonstrate the reaction of our malleable version of Tensorflow against resource modification requests, we illustrate different examples of modifications of intra- and inter-parallelism on a long-run training procedure for Resnet model. Specifically, Figure 4 reports three different examples of resource usage modifications. This example considers that the modification of the degree of parallelism is performed between training epochs, although it can be requested at any execution time. In the figure, we include the exact points in which parallelism is modified and how the execution time for the next epochs increases or decreases instantly.

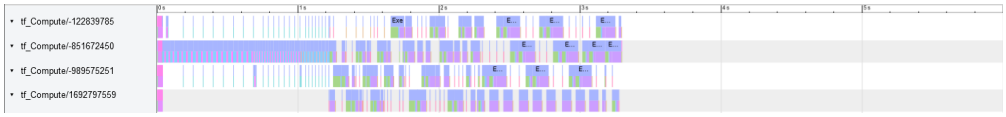
The Tensorboard component in Tensorflow can also be used to illustrate modifications in inter- and intra-parallelism. In Figure 5 we illustrate the modification and execution times of different phases of the training procedure when modifying inter-parallelism (left traces) and intra-parallelism (right traces). These results validate the dynamic nature of the elasticity method introduced in Tensorflow.

6.4 Scheduling and resource management policies

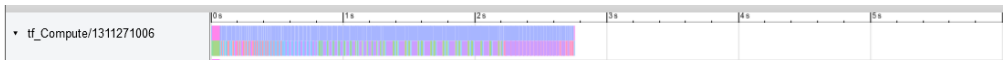
The following experiments aim at illustrating the potential benefits of coupled elasticity-aware scheduling for different workloads. Figure 6 reports the observed results for TME, TMR and TMER for the three resource re-assignment policies (Strict, Always_Attend and Max_Prop) for different workload profiles (resource pressure is reduced through scenarios and decreasing from the top plots to the bottom ones) using an increasing amount of concurrent task creation requests (from 8 to 128), and combined with the original Tensorflow version, and our malleable Tensorflow implementation. The plot also includes the observed resource usage (core occupation percentage) for each combination. In these experiments, we have used

(a) Stage 1: Inter=1. Time: 4.9s.

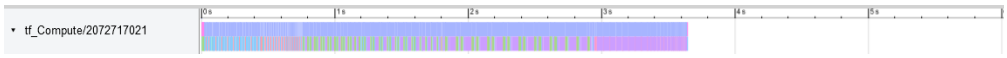
(b) Stage 1: Intra=1. Time: 4.6s.



(c) Stage 2: Inter=4. Time: 3.1s.



(d) Stage 2: Intra=4. Time: 2.8s.



(e) Stage 3: Inter=1. Time: 3.1s.

(f) Stage 3: Intra=1. Time: 4.4s.

Figure 5: Tensorboard traces for INTER parallelism (subfig. a,b and c) and INTRA parallelism (subfig. d, e and f) manipulation.

an FCFS scheduling policy. Similarly, Figure 7 reports a comparative view of the malleable version of Tensorflow compared with a standard version, in terms of time improvement (comparative results are only reported for high resource usage scenarios); the figure also reports the improvement in productivity after the execution of all tasks. A number of insights can be extracted:

- **Impact of the workload intensity.** Workload intensity (or resource pressure) can be obtained in our setup by modifying two parameters: the number of concurrent containers, and the amount of requests performed (workload definition). Figure 7 gives a clear indication of its impact on performance in combination with coupled elasticity. Scenarios with high request ratio and with a higher number of concurrent containers, reveal the best time improvements from the elastic approach compared with a static one, yielding improvements of up to 80% in time.

- **Impact of coupled elasticity.** In all cases in which the requests pressure is high, the malleable version of Tensorflow achieves remarkable accelerations compared with its static counterpart. For workloads with low resource requests, the benefits are less evident or imply small performance penalties.

- **Impact of the re-assignment policy.** The elastic version of Tensorflow exhibits benefits with ALWAYS_ATTEND and MAX_PROP. Recall that combining these policies with a static resource management can yield oversuscription or under-utilization effects. MAX_PROP, specifically combined with a coupled elasticity management, yields in general the best resource utilization and hence execution times.

Finally, Fig. 8 summarizes, in terms of TMER, the impact of a priority-based scheduling scheme in the final execution time of tasks with different priorities (in the figure, a priority level 0 corresponds to the highest priority). The figure reports the average TMER for all tasks corresponding to a specific priority level, generated randomly. The insights here are two-fold; first, there is a clear impact (improvement) in the execution time for those tasks with higher priority; second, the improvement compared with the original Tensorflow implementation remains similar to that observed for FCFS.

6.5 Portability across models

Finally, Figure 9 shows the portability of our solution in different models. Specifically, we report the productivity improvement of the overall elastic scheduling for three different models (ResNet, EfficientNet, and ConvNet) as described in Section 6.2. The results are reported in terms of improvement in productivity compared with the original Tensorflow implementation. Observe how, specially for experiments that exhibit

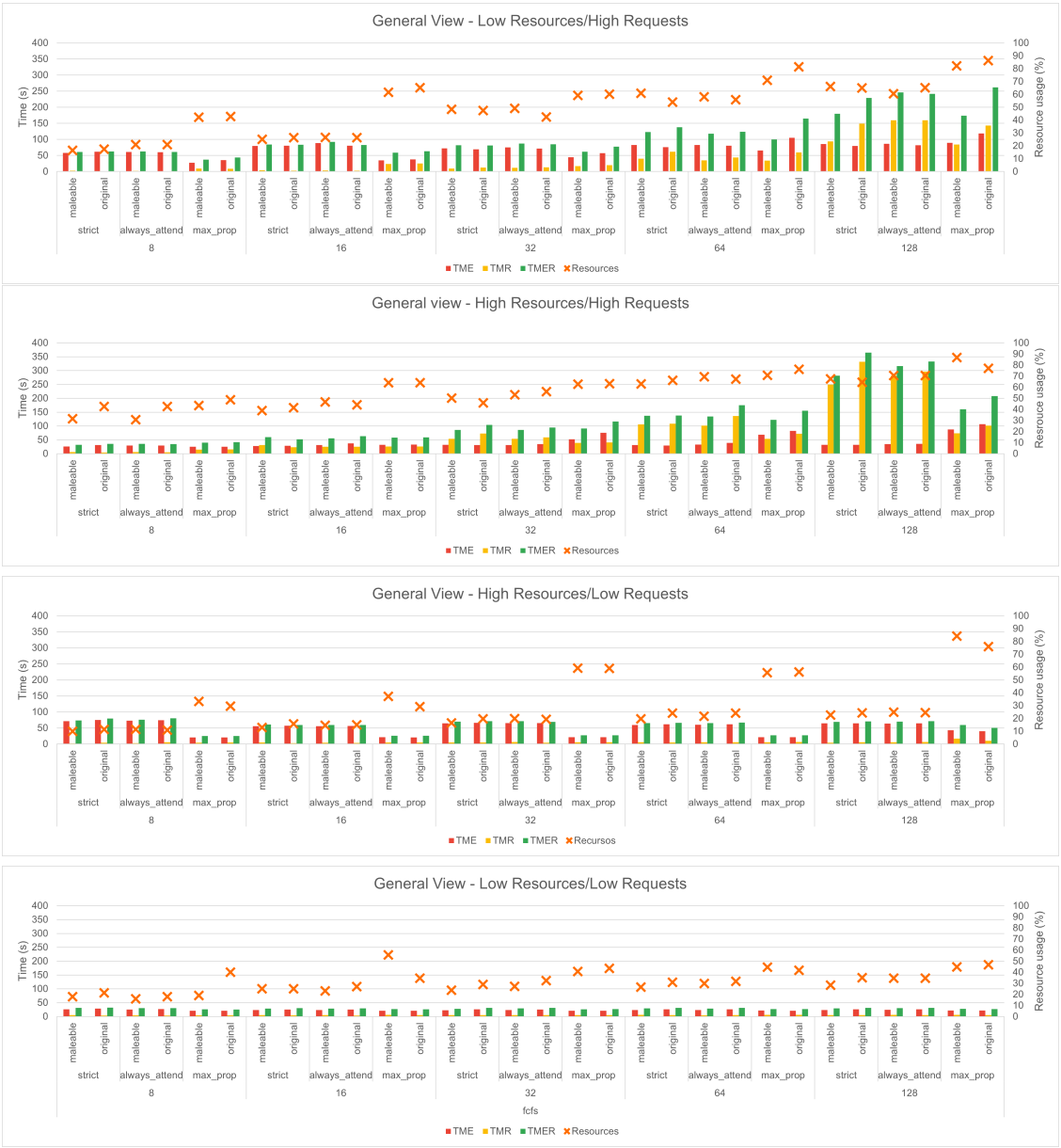


Figure 6: Time metrics and resource usage for different workloads on the testbed server.

high request rates, the behavior for the three models is qualitatively similar, which reinforces and validates the portability of the solution across different workloads.

7 Conclusions and future work

In this paper, we have proposed a coupled elasticity-aware scheduling mechanism of applications confined within containers, in which resource management is performed simultaneously at two levels: at the container level, leveraging the mechanisms offered by Docker to assign and re-assign resources (cores) to containers; and at the application level, proposing Tensorflow as a target application to illustrate the development of a malleable application. The observed performance results reveal that this two-tier elasticity strategy is mandatory to avoid oversuscription and under-utilization of resources at the application level, while maintaining a proper core occupation across a modern multi-core server.

Future work includes the exploration of different scheduling policies such as Round-Robin and Sporadic, including the use of priority and preemption. Also, introducing application heterogeneity by using different convolutional network training models and extend to transformer-based models such as GPT or Bert is of our interest in order to further extend the portability study. One of the most important future directions is to include elastic GPU scheduling in Tensorflow or similar framework like PyTorch using the Multi-

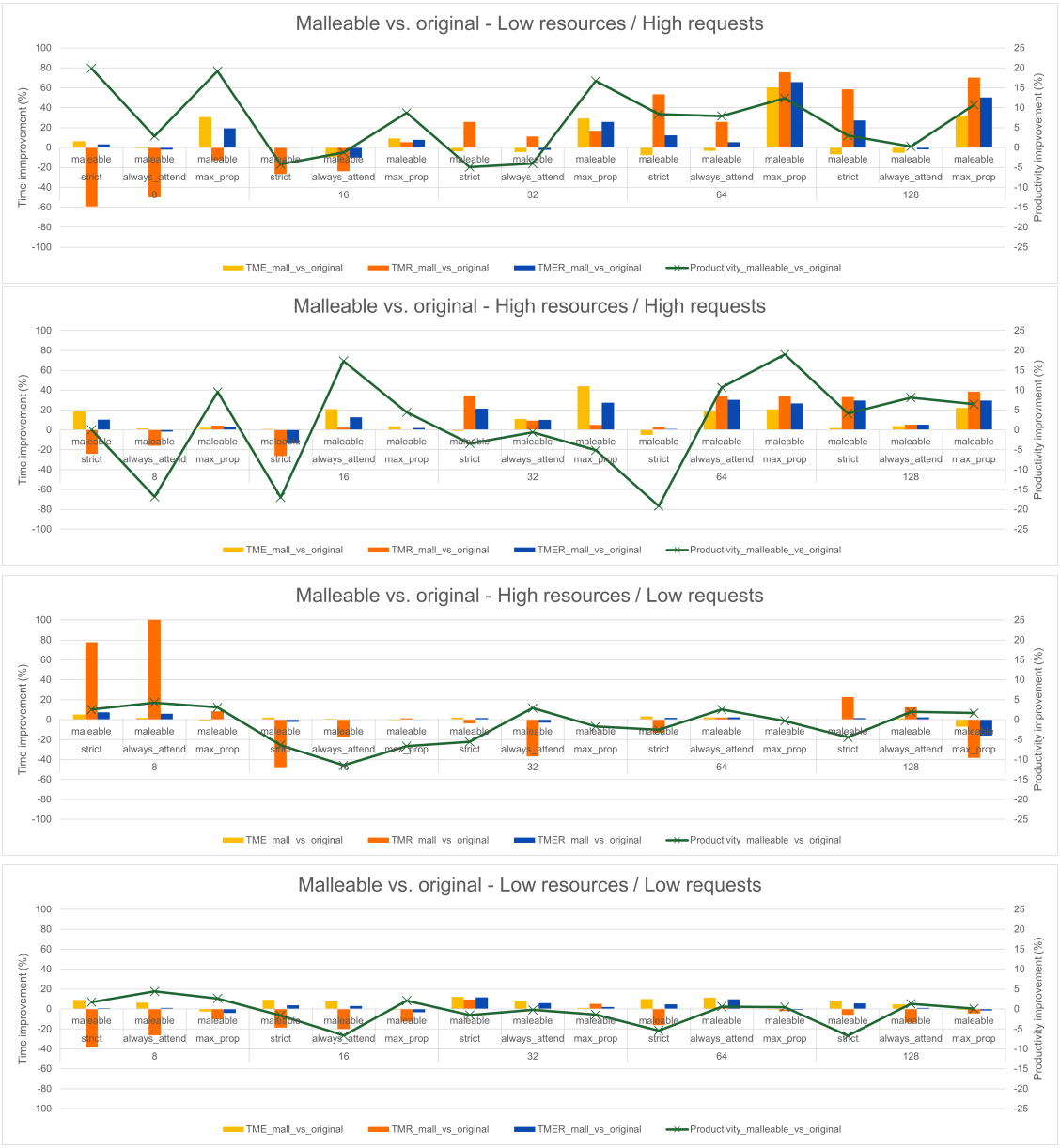


Figure 7: Comparative metrics (for time and productivity) for different workloads on the testbed server.

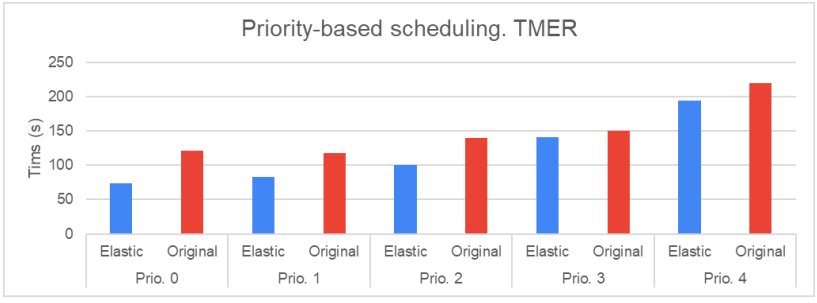


Figure 8: Execution times for the priority-based scheduler.

Instance GPUs (MIG) technology present in the latest generations of NVIDIA GPUs that allows reconfiguring resources to avoid under-subscription of resources.

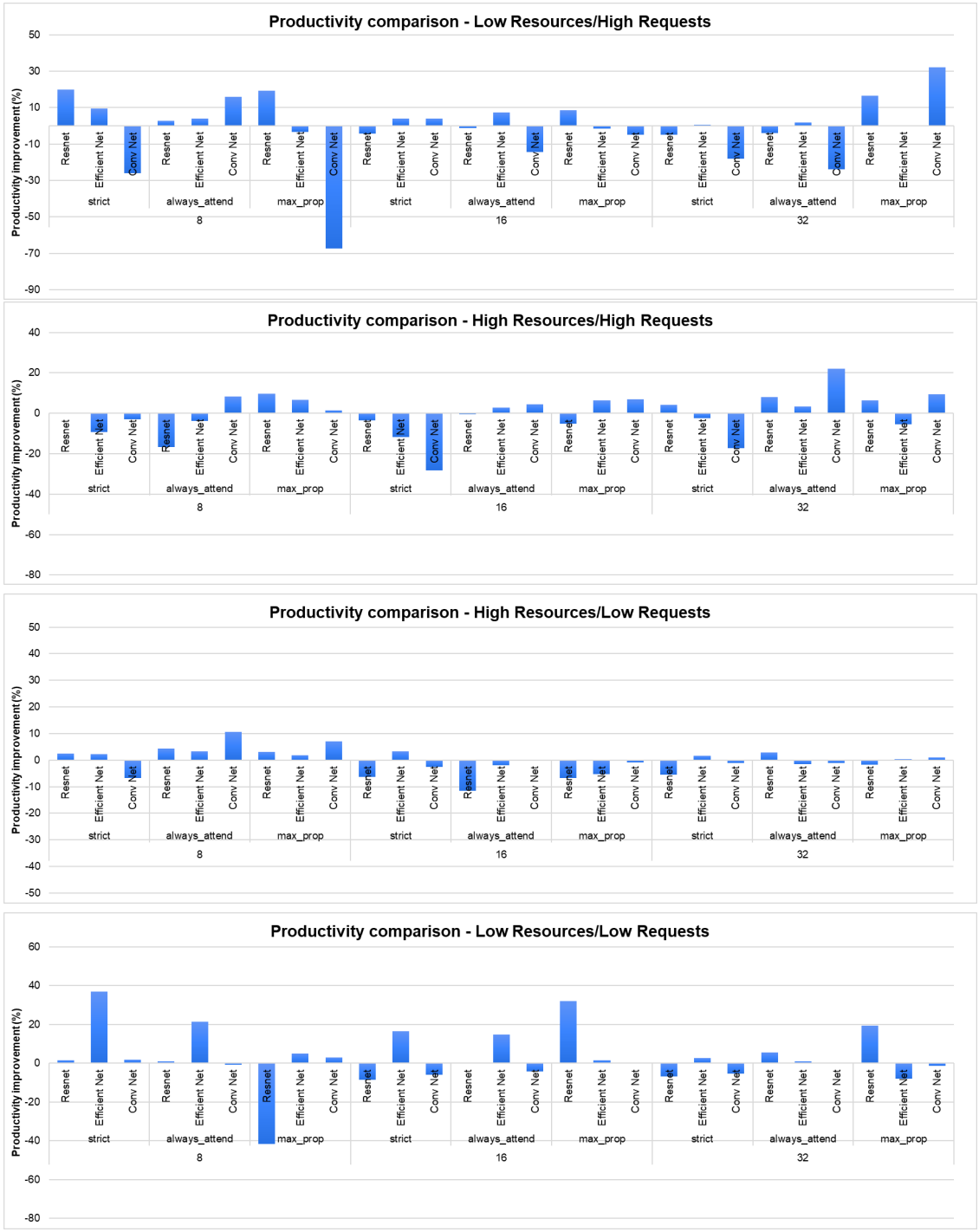


Figure 9: Productivity comparison across models.

References

[1] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 329–338.

[2] E. Casalicchio, "Container orchestration: A survey," *Systems Modeling: Methodologies and Tools*, pp. 221–235, 2019.

[3] S. Abraham, A. K. Paul, R. I. S. Khan, and A. R. Butt, "On the use of containers in high performance computing environments," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 2020, pp. 284–293.

[4] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 82–89.

[5] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *2017 IEEE 10th international conference on cloud computing (CLOUD)*. IEEE, 2017, pp. 472–479.

[6] E. Preeth, F. J. P. Mulerickal, B. Paul, and Y. Sastri, "Evaluation of docker containers based on hardware utilization," in *2015 international conference on control communication & computing India (ICCC)*. IEEE, 2015, pp. 697–700.

[7] H. T. Ciptaningtyas, B. J. Santoso, and M. F. Razi, "Resource elasticity controller for docker-based web applications," in *2017 11th International Conference on Information & Communication Technology and System (ICTS)*. IEEE, 2017, pp. 193–196.

[8] Y. H. Oh, S. Kim, Y. Jin, S. Son, J. Bae, J. Lee, Y. Park, D. U. Kim, T. J. Ham, and J. W. Lee, "Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 584–597.

[9] W. Zheng, M. Tynes, H. Gorelick, Y. Mao, L. Cheng, and Y. Hou, "Flowcon: Elastic flow configuration for containerized deep learning applications," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[10] Docker. [Online]. Available: https://www.docker.com/

[11] R. Doukha, S. A. Mahmoudi, M. Zbakh, and P. Manneback, "Deployment of containerized deep learning applications in the cloud," in *2020 5th International Conference on Cloud Computing and Artificial Intelligence: Technologies and Applications (CloudTech)*. IEEE, 2020, pp. 1–6.

[12] Docker swarm. [Online]. Available: https://docs.docker.com/engine/swarm/

[13] Kubernetes. [Online]. Available: https://kubernetes.io/es/

[14] Marathon. [Online]. Available: https://mesosphere.github.io/marathon/

[15] Apache mesos. [Online]. Available: https://mesos.apache.org/

[16] Cloudify. [Online]. Available: https://cloudify.co/

[17] H. Zhao, W. Cui, Q. Chen, J. Leng, K. Yu, D. Zeng, C. Li, and M. Guo, "Coda: Improving resource utilization by slimming and co-locating dnn and cpu jobs," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 853–863.

[18] J. Mohan, A. Phanishayee, J. Kulkarni, and V. Chidambaram, "Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 579–596.

[19] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–14.

[20] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 595–610.

[21] J. Han, M. M. Rafique, L. Xu, A. R. Butt, S.-H. Lim, and S. S. Vazhkudai, "Marble: A multi-gpu aware job scheduler for deep learning on hpc systems," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 272–281.

[22] S. Wang, O. J. Gonzalez, X. Zhou, T. Williams, B. D. Friedman, M. Havemann, and T. Woo, "An efficient and non-intrusive gpu scheduling framework for deep learning training systems," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–13.

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