Dyna-P: placement-aware dynamic partitioning for lightweight applications with modern GPUs

Dyna-P: Placement-aware Dynamic Partitioning for Lightweight Applications with Modern GPUs

Theodora Adufu[1] and Yoonhee Kim[2*]

[1,2]Department of Computer Science, Sookmyung Women's University, 100 Cheongpa-ro 47-gil, Seoul, 04310, South Korea.

*Corresponding author(s). E-mail(s): yulan@sookmyung.ac.kr;
Contributing authors: theoadufu@sookmyung.ac.kr;

Abstract

Efficient GPU resource sharing is critical in dynamic cloud-based environments, particularly for lightweight HPC applications and Small Language Models, which demand partial GPU resources for execution. However, traditional scheduling frameworks fail to address intra-GPU and inter-node resource fragmentation and dynamic placement challenges arising from the heterogeneity in each application's resource demand and job completion times. This leads to resource under-utilization and scheduling delays in GPU clusters. This paper introduces Dyna-P, a novel scheduling framework designed to dynamically adjust GPU partitions to minimize resource fragmentation while improving system throughput and Makespan. Dyna-P proposes a Reconfiguration Last Placement policy which recognizes that workloads consisting of lightweight applications can benefit more from uninterrupted execution. Experimental results demonstrate that Dyna-P improves average throughput by up to 14.7% and reduces Makespan by 39% compared to state-of-the-art methods. These findings underscore Dyna-P's potential to improve resource allocation rates in multi-tenant GPU environments.

Keywords: Dynamic Partitioning, Spatial Sharing, GPU utilization, Placement, Fragmentation

1 Introduction

Modern Graphics Processing Units (GPUs) are pivotal in accelerating workloads across diverse fields, including Artificial Intelligence (AI), High-Performance Computing (HPC), and scientific research. The increasing prevalence of cloud-based GPU environments [1, 2, 3, 4] and the recent introduction of local inference serving platforms like Ollama [5] have driven the need for efficient resource management, particularly as workload diversity continues to expand. Container orchestrators like Kubernetes and KubeEdge[6, 7] play a vital role in workload scheduling within resource-constrained GPU cloud environments, such as AI research institutes and edge servers. However, these systems face significant challenges in efficiently managing GPU resources, particularly for lightweight applications. These challenges are intensified by the rapid adoption of Small Language Models (SLMs), which introduce diverse and unpredictable resource demands, and by the irregular arrival of jobs, which complicates scheduling and allocation strategies.

Similar to some HPC applications [8, 9], SLMs, with their lightweight architecture and high efficiency, often require partial GPU resources, creating a complex multidimensional bin-packing problem [10, 11] with placement constraints; a

challenge yet to be fully addressed by existing schedulers.

Existing GPU sharing methodologies can be broadly classified into temporal and spatial sharing techniques. Temporal sharing, which involves alternating GPU access across time partitions, often leads to performance degradation as a result of frequent context switching. In contrast, spatial sharing, enabled by technologies such as NVIDIA's Multi-Instance GPU (MIG), partitions GPU resources into isolated partitions to support concurrent execution. However, these approaches are limited by static configurations, resource fragmentation, and inefficiencies in addressing dynamic job requirements.

This paper proposes Dyna-P, a dynamic partitioning framework designed to improve GPU resource allocation rates for lightweight applications. Using inherent *merge* and *split* functionalities, Dyna-P dynamically reconfigures partial GPU partitions to minimize fragmentation and maximize resource utilization. Dyna-P also proposes a Reconfiguration Last Placement policy to ensure uninterrupted execution of workloads. Experimental evaluations highlight the following key contributions:

- Analysis of lightweight applications on GPU partitions to estimate required resources and address over-provisioning (Sec. 2).
- Innovative use of NVIDIA's *merge* and *split* features for flexible GPU configurations and minimization of intra-GPU fragmentation.
- A joint resource selection, placement, and scheduling algorithm designed to enhance throughput and minimize makespan by leveraging spatio-temporal workload characteristics and placement awareness (Sec. 3).

The rest of the paper is organized as follows: Section 2 highlights the motivations for this investigation. Section 3 describes the proposed Dyna-P framework. Section 4, presents an evaluation of Dyna-P through extensive experiments. Section 5 reviews related work, while Section 6 discusses potential applications. Finally, Section 7 concludes the paper.

2 Background and Motivation

In this section, scheduling in GPU cluster environments is analyzed using Alibaba's GPU cluster trace from 2023 [12] to highlight the scheduling challenges yet to be addressed by existing schedulers.

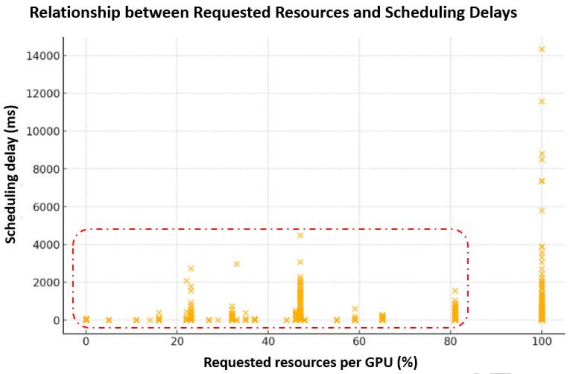


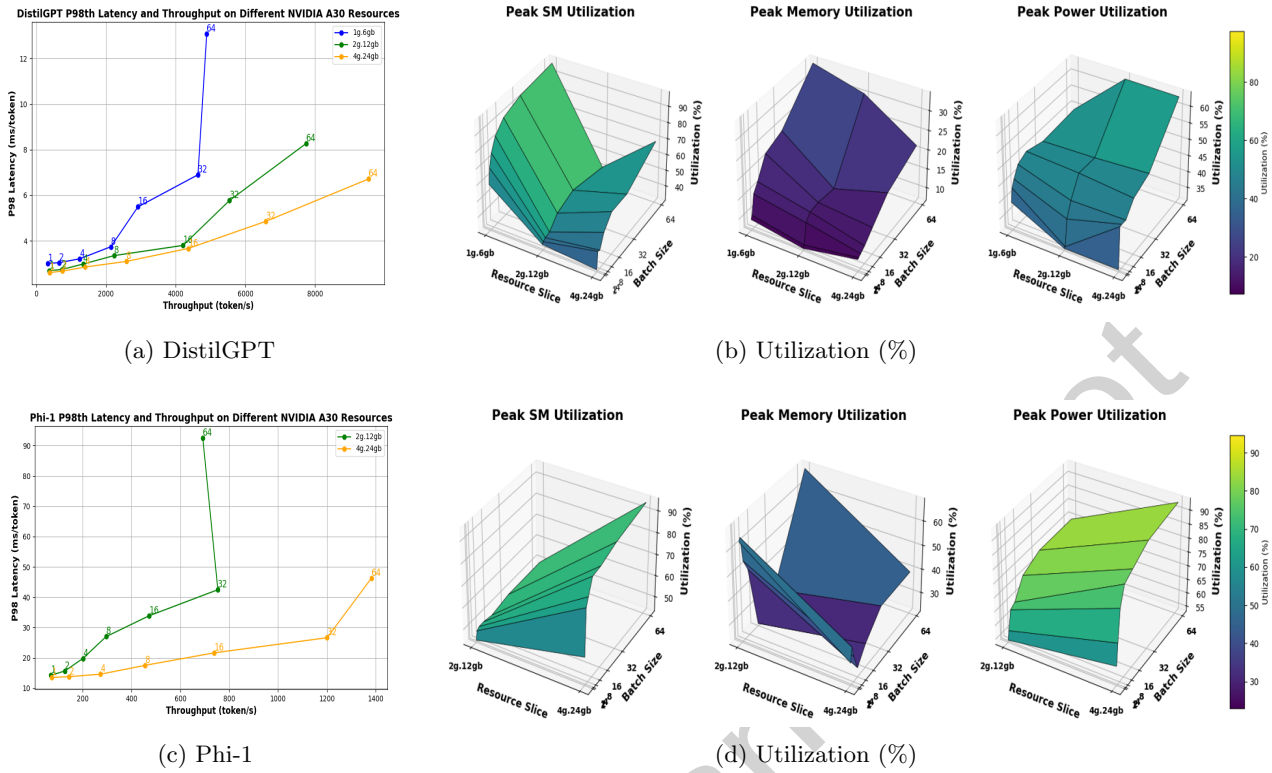
Fig. 1: Relationship between requested resources and scheduling delays using Alibaba's GPU cluster trace 2023[12]

As observed in Figure 1, consistent scheduling delays are not only linked to irregular job arrival patterns but also to the high variability in job resource requirements. Workloads requiring small GPU partitions often experience delays (Figure 1) as traditional GPU runtime environments assign full GPU resources to jobs which fail to fully utilize them. This observation has led to a surge in research focused on improving GPU utilization [13, 14, 15, 16]. However, these do not consider placement sensitivity and strategies to harvest resource fragments to improve resource allocation rates and hence job queuing times. Research into GPU sharing methodologies has focused on temporal and spatial sharing(fine-grained and coarse-grained), with the latter showing promise in improving GPU utilization.

Fine-grained GPU sharing [17, 18, 19] enhances hardware utilization at the SM or Compute Unit (CU) level, thereby improving overall system throughput. For instance, NVIDIA's Multi-Process Service (MPS) [17] allows multiple kernels to share GPU resources. However, fine-grained sharing introduces challenges, including interference among concurrently running workloads, where kernels may modify overlapping memory locations. Additionally, the lack of strict performance isolation makes MPS unsuitable for multi-tenancy in GPU clusters.

Coarse-grained approaches, such as NVIDIA's Multi-Instance GPU (MIG) [20] and

2



(a) DistilGPT (b) Utilization (%)

(c) Phi-1 (d) Utilization (%)

Fig. 2: Performance comparison SLM workloads for different GPU configurations

AMD's Compute Unit Masking [21], enable resource partitioning into isolated partitions to support concurrent execution. For NVIDIA's MIG, resource partitions are denoted by profiles like 4g.24gb, where 4g represents the compute capacity and 24gb refers to the associated memory allocation. MIG supports limited levels of concurrency, allowing up to seven workloads to run concurrently on architectures like A100 or H100. However, these configurations are static and changing them to accommodate new jobs requires stopping all active jobs, leading to inefficiencies and delays [11]. NVIDIA's *merge* and *split* features address some limitations by enabling the creation of new partitions from existing ones without requiring full reconfiguration. However, these operations are constrained by the physical locations of available partitions and the placement of active jobs, making dynamic scheduling in online environments particularly challenging.

In multi-tenant GPU clouds, coarse-grained spatial-sharing approaches like NVIDIA's MIG are often preferred due to resource isolation. However, determining suitable partitioning and resource allocation for inference jobs remains a challenge; workloads are heterogeneous in their resource demands.

In the remainder of this section, analysis on resource sharing, utilization, and throughput for lightweight applications is conducted to address the following research questions.

- How can GPU partitions be effectively allocated to workloads? (Sec. 2)
- What configurations and placements best enhance performance for diverse workloads? (Sec. 3)
- Which scheduling strategies improve resource utilization and reduce fragmentation?(Sec. 3)

2.1 Resource Density, Scalability and Performance

In this section, the performance of a specific group of lightweight applications known as SLMs

3

is evaluated for performance on different resource partitions.

According to [22], SLMs with parameters between 0.1 Billion and 3 Billion require between 275 MB and 2456 MB of memory. Their modest resource needs and efficiency in handling tasks such as code generation, summarization, and text classification have driven their adoption in modern GPU clouds. The allocation of Streaming Multiprocessors (SMs), memory bandwidth, and cache memory is thus critical to improving delays and throughput[23] especially for batch inference requests.

Figure 2 illustrates the execution of two representative inference jobs, DistilGPT [24] and Phi-1 [25], to evaluate how resource density (allocated GPU resources) impacts SLM performance and GPU resource utilization. Using NVIDIA's MIG, isolated partitions of 1g.6gb, 2g.12gb, and 4g.24gb on the NVIDIA A30 GPU were provisioned for this investigation. Performance metrics, including model throughput and P98th latency, were measured for each workload across batch sizes (1, 2, 4, 8, 16, 32, 64).

In Equation 1, model throughput (MT) measures the total number of input and output tokens processed per second. P98th latency represents the time required to complete 98% of inference requests for a given batch size. In these benchmark experiments, inference requests with the same input token length were executed 10 times for each batch size and MIG partition.

$$\text{MT} = \frac{\text{Total tokens (Input + Output)}}{\text{Inference time}} \quad (1)$$

From Figure 2, it is observed that the performance of both applications deteriorates sharply beyond a trade-off point (TP) on each partition. Figure 2 shows that P98th latency increases significantly for smaller partitions (1g.6gb) compared to the full GPU (4g.24gb). This is due to resource constraints in smaller partitions, which increase kernel launch times, computation delays, and data fetching overheads, especially for larger batch sizes.

In particular, DistilGPT shows higher throughput than Phi-1 and less latency sensitivity between batch sizes. In contrast, Phi-1 experiences a steep increase in latency and a fall in model throughput between batch sizes 32 and 64 on the 1g.6gb partition, highlighting its higher demand for compute resources and the limitation posed by slower memory operations. Furthermore, Phi-1's failure to execute on the 1g.6gb partition underscores the challenges of resource constraints, while DistilGPT's low resource utilization on 2g.12gb highlights the challenge of balancing under-provisioning and over-provisioning during scheduling. Additionally, peak power utilization increases with batch size and partition sizes for both applications however, the higher power usage for DistilGPT on 2g.12gb shows that selecting the right resource partition ultimately affects the energy efficiency and carbon footprints when serving inference.

Takeaway: *Dynamic input parameters such as batch sizes, affect the throughput and latency of SLM inference across different partitions. Executing applications with larger batch sizes on any partition is less beneficial beyond a trade-off point.*

2.2 Placement Sensitivity and System Throughput

The effect of job placement preferences in improving performance has been studied for scheduling scenarios[26, 27, 28] with heterogeneous resources.

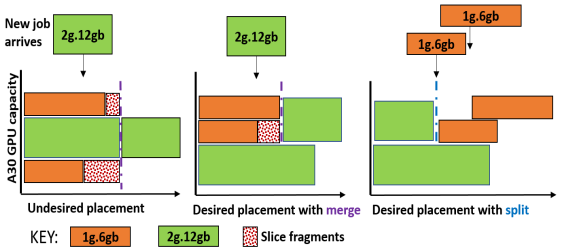
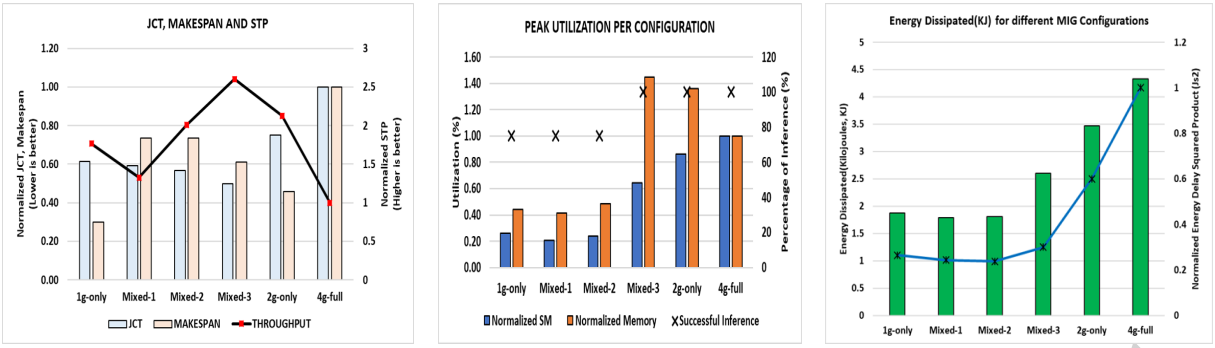


Fig. 3: Desired spatio-temporal placement Scenarios with *merge* and *split*

From previous work [8] and as depicted in (Figure 3), certain valid GPU configurations may not be suitable for a given workload, despite the logical availability of resources within a given scheduling period. This phenomenon or **undesired placement** is usually observed as systems adapt to fluctuating incoming inference requests when resources are scaled dynamically. This usually leads to resource fragmentation[12] and idle GPU time.



(a) JCT, Makespan, STP (b) Resource utilization (c) Total energy dissipated

Fig. 4: Performance comparison of workloads for different GPU configurations

NVIDIA's MIG sharing provides *merge* and *split* features, which allow the dynamic creation of new resource profiles to meet user QoS requirements without preempting all active jobs. While these features can reduce the total time required to complete all jobs (makespan) and improve system throughput, their effectiveness is limited by architectural constraints (Figure 3). When leveraged appropriately, they can improve resource allocation and significantly enhance performance.

In multi-GPU cluster environments, inference workloads are diverse in their QoS requirements, arrival rates, model throughput (Sec. 2.1), and utilization across different GPU resources. This analysis explores the impact of placement strategies on system throughput during the concurrent execution of inference jobs across various MIG configurations. By understanding how placement sensitivity affects resource utilization, this work aims to inform strategies for balancing workload demands and maximizing system performance.

In Figure 4, an NVIDIA A30 GPU is statically partitioned into six configurations: (**1g-only:** 1g.1g.1g.1g, **Mixed-1:** 1g.1g.2g, **Mixed-2:** 1g.2g.1g, **Mixed-3:** 2g.1g.1g, **2g-only:** 2g.2g, and **4g-full:** 4g). These configurations consist of both homogeneous and heterogeneous (mixed) partitions, with jobs assigned to GPU partitions using a first-fit placement strategy. The inference workloads of four SLM models—DistilGPT [24], Phi-1 [25], CodeGen [29] and Flan-T5-Large [30]—arrive in a First-Come-First-Served (FCFS) order for execution.

Sequel to prior studies [9, 31, 32], the system throughput (Equation 5 in Sec. 3.1.2) is calculated as the weighted sum of the relative performance of each GPU partition compared to a full GPU resource. From Figure 4b, it is observed that without GPU reconfiguration, the placement of jobs for varying inference workloads leads to different system throughput and job completion times across configurations. Due to the variation in minimum resource requirements, partition allocations for certain jobs, such as Phi-1, result in job failures in some configurations. This is evident in the 1g-only configuration, where the 1g.6gb partition cannot satisfy Phi-1's QoS requirements, leading to execution failures.

Takeaway: *Minimizing the trade-off between system performance degradation and resource utilization requires a scheduler that anticipates job placement outcomes across diverse use-case scenarios in multi-tenant GPU clouds.*

3 Placement-aware GPU partitioning: Dyna-P

Dyna-P is a workload and placement aware spatial sharing system designed to enhance GPU resource utilization and reduce scheduling delays caused by resource fragmentation. To achieve these objectives, the scheduling framework must account for both spatial and temporal requirements of workloads. Spatial requirements involve allocating appropriate GPU partition sizes and ensuring efficient utilization, while temporal requirements address the dynamic nature of workloads, including job arrivals, completions, and GPU resource availability over time.

5

Dyna-P comprises two main components: **Capacity Evaluator** and **Scheduling Unit**, and operates as described in the example scenario below (Figure 5). The architecture is designed to dynamically adapt GPU partitions and schedule inference workloads in multi-tenant environments so as to maximize system throughput while minimizing resource fragmentation and scheduling delays.

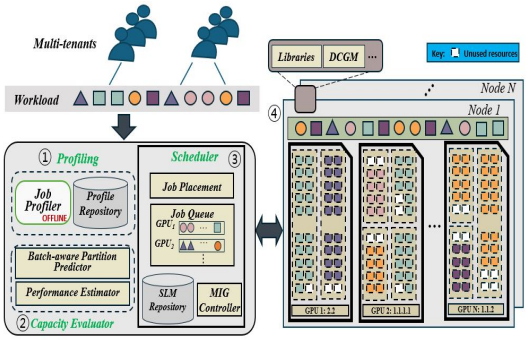


Fig. 5: System Design of Dyna-P

In a multi-GPU environment, multiple tenants submit jobs, with varying characteristics to a shared queue, $Q = (a\_b_j, \ldots, a\_b_m)$. During workload submissions, users specify batch size b and QoS requirements such as deadline ($D_{a\_b_j}$), while also indicating that resources can be shared among workloads. Users sometimes submit the same jobs with different batch sizes; this variance in batch sizes is represented by b in our nomenclature. For this scenario, the main procedure of Dyna-P is summarized as follows:

1. In an offline process, each submitted inference job is profiled in a FCFS manner. The profiling process records important characteristics of the job, which are stored in a profile repository, $P(p_{a\_b_j}, ..., p_{a\_b_m})$.
2. Using the workload profiles, the Batch-aware Partition Predictor, considers the batch size, b, of the inference job and addresses the issue of resource over-provisioning by ensuring that jobs receive resources which improve throughput at low latencies. The Job Completion Time of each job, $JCT_{a\_b_j, g_i}$, is then estimated using the performance counters in the profiles.
3. Jobs are prioritized based on their QoS requirements ($D_{a\_b_j}$) and the branch-and-bound algorithm is called to determine the most suitable placement on each local GPU subject to users' deadline constraints.
4. During assignment, Dyna-P continuously monitors the workload queue and GPU states. and evaluates the GPU for potential resource merges or splits when the physical partitions differ from the partition required for each placement. This step accounts for job arrival patterns and anticipates upcoming workload requirements by leveraging workload profiles.This process repeats until the workload queue is empty.

By proactively reallocating resources based on estimated needs, Dyna-P minimizes delays and improves scheduling efficiency.

3.1 Capacity Evaluator

Analyzing and allocating GPU resources to inference jobs is critical, as performance outcomes heavily depend on the behavior of each workload. However, users often oversubscribe to GPU clusters [33] in an attempt to improve execution outcomes, which paradoxically leads to resource under-utilization. Dyna-P leverages NVIDIA's Data Center GPU Manager (DCGM) profiling tool [34] to collect job execution and utilization metrics for batch inference jobs. With job profiles, the capacity evaluator predicts the most suitable resource partition and the expected job completion times for each job, as described below.

3.1.1 Batch-aware Partition Predictor

Dyna-P supports multi-GPU environments with MIG-enabled GPUs, $G = (G_1, \ldots, G_k)$, allowing for both homogeneous and heterogeneous partitioning. Each partition g_i represents a fraction of a GPU's resources. It is assumed that inference jobs require either partial or full GPU resources, with g_i satisfying $0 < g_i \leq G_k$ where $G_k = (g_1, \ldots, g_{|G|})$, the set of valid GPU partitions. The maximum number of valid partitions vary per GPU architecture; for instance on the A30 GPU, $|G|= 4$ whilst $|G|= 7$ for A100 and H100 GPUs.

As illustrated in Figure 2, each inference job shows a latency (Lat) and model throughput (MT) trade-off for each combination of batch sizes

and resource partitions. This shows that latency and model throughput must both be considered simultaneously in order to select suitable resources and ensure that user's QoS (deadline) are satisfied. It is desirable that latency be at its minimum while model throughput improves.

The Batch-aware Partition Predictor adapts Li et al. [35]'s use of directed bipartite graphs to match inference jobs of varying batch sizes a_b_j to suitable GPU partitions g_i, where a_b_j denotes job a_b_j executed with user-specified batch size b. Dyna-P's graph-based approach efficiently models the trade-offs between latency, model throughput, and resource partitions. In the graph, one set of nodes represents batch size variants a_b_j, and the other set represents available GPU partitions g_i. A weighted edge, $F(a\_b_j, g_i)$, between a_b_j and g_i represents the suitability of that partition for executing the job with weights calculated using Equation 2.

$$F(a\_b_j, g_i) = \lambda \cdot \Delta Lat_{a\_b_j, g_i} + (1-\lambda) \cdot \Delta MT_{a\_b_j, g_i} \quad (2)$$

$$s.t. \quad Lat(a\_b_j, g_i) + margin \leq D_{a\_b_j} \quad \forall g_i, a\_b_j$$

The weights balance latency minimization ($\Delta Lat_{a\_b_j, g_i}$) and model throughput maximization ($\Delta MT_{a\_b_j, g_i}$) based on the parameter λ where $0 \leq \lambda \leq 1$ is a configurable parameter that can be tuned for different scenarios. The baseline metrics $L_{base_{g_i}}$ and $MT_{base_{g_i}}$ in Equations 3 and 4 are measured for the maximum batch size (e.g. $b = 64$) that the partition g_i can accommodate. This represents the most resource-intensive scenario and ensures that weights are normalized for consistent comparisons.

$$\Delta Lat_{a\_b_j, g_i} = \frac{L_{base_{g_i}} - L(a\_b_j, g_i)}{L_{base_{g_i}}} \quad (3)$$

$$\Delta MT_{a\_b_j, g_i} = \frac{MT(a\_b_j, g_i) - MT_{base_{g_i}}}{MT_{base_{g_i}}} \quad (4)$$

Using this representation, BPP (Algorithm 1) performs matching by identifying (a_b_j, g_i) pairings that maximize edge weights, prioritizing partitions that reduce latency and maximize model throughput, while ensuring that QoS deadlines $D_{a\_b_j}$ are met. During the graph construction phase, edges are created only if the memory and compute requirements of a_b_j do not exceed the capacities of g_i. If the memory requirements of the job do not exceed the partition capacity ($Mem_{a\_b_j} \leq Cap_{g_i}$), an edge is created, with its weight $F(a\_b_j, g_i)$ calculated based on Equation 2. Invalid pairs are excluded at this stage to ensure that only feasible assignments are considered.

Algorithm 1 Edge Creation with BPP

Require: Jobs $Q = a\_b_1, a\_b_2, \ldots, a\_b_m$
Require: Partitions $G_k = g_1, g_2, \ldots, g_{|G|}$
for job $a\_b_j \in Q$ **do**
  for partition $g_i \in G_k$ **do**
    if $Mem_{a\_b_j} \leq Cap_{g_i}$ **then**
      $F(a\_b_j, g_i) \leftarrow \lambda \cdot \Delta Lat_{a\_b_j, g_i} + (1 - \lambda) \cdot \Delta MT_{a\_b_j, g_i}$
      Add edge $(a\_b_j, g_i, F(a\_b_j, g_i))$
    else
      Exclude (a_b_j, g_i)
    end if
  end for
end for

Suppose there are two inference jobs, a_b_1 and a_b_2, with batch sizes $b = 16$ and $b = 32$, respectively. The available GPU partitions are 1g.6gb, 2g.12gb and 4g.24gb and they all satisfy the OOM requirements of a_16_1, but 1g.6gb does not meet the requirements of a_32_2. Then using Equation 2, weights $F(a\_b_j, g_i)$ are calculated for each valid pairing as illustrated in Figure 6. Edges

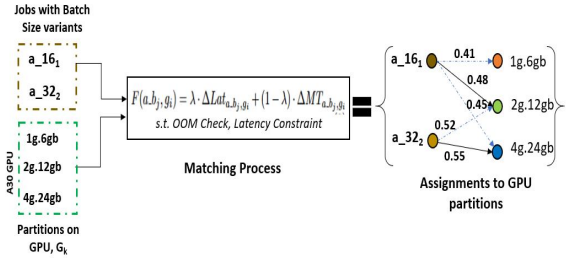


Fig. 6: Batch-aware Partition Prediction Process

are created in the bipartite graph for these pairs, ensuring that 2g.12gb and 4g.24gb are prioritized for a_32_2, while 1g.6gb, 2g.12gb and 4g.24gb remain feasible for a_16_1. The weighted edge with the highest $F(a\_b_j, g_i)$ value is considered the most suitable partition for the job. If multiple partitions yield the same $F(a\_b_j, g_i)$, the smallest slice

is selected as the tie-breaker to maximize resource efficiency.

3.1.2 Performance Estimator: System Throughput

The temporal aspect of scheduling is crucial for improving system throughput in multi-tenant environments. In Dyna-P, the performance estimator determines the system throughput of workloads submitted by multi-tenants. With performance counters collected for the application during profiling, Dyna-P uses XGBoost regression model [36] to estimate the job completion time for each inference job on its predicted partition ($JCT_{a\_b_j,g_i}$).

System throughput, T_{G_k}, quantifies the efficiency of allocated GPU partitions, g_i in processing inference jobs, a_b_j, while meeting user-defined QoS constraints. Following prior work [9, 32, 37], T_{G_k} is calculated as the weighted sum of the relative throughput of the inference workloads, $RT_{a\_b_j,g_i}$, on each partition. Formally:

$$T_{G_k} = m_{G_k} \cdot \sum_{i=1}^{i<W} RT_{a\_b_j,g_i} \tag{5}$$

where $RT_{a\_b_j,g_i}$ is the speed-up in of inference job (a_b_j) executed on full GPU relative to its performance on a partition g_i. To account for idle GPU partitions and ensure that the system throughput reflects the actual utilization of the GPU for any combination of concurrently executed jobs, the weight, m_{G_k}, is introduced. m_{G_k} is the proportion of the GPU resources used during any concurrent execution of inference jobs relative to the capacity of the whole GPU. The parameter W represents the number of inference jobs executed concurrently on a single GPU.

3.2 Scheduling Unit

Efficient resource allocation in private clusters and multi-GPU environments is essential for minimizing makespan (the total job execution time) and ensuring timely processing of new inference submissions. Each GPU schedules jobs based on their resource demands, available capacity (R_{avail}), and inherent architectural and execution constraints. This subsection discusses the constraints, allocation strategies and job placement considerations.

Key notations used in the scheduling unit are summarized in Table 1.

Table 1: Notations

Notation	Remarks
Q	Workload queue with submitted inference jobs, $Q = (a\_b_j, ..., a\_b_m)$
a_b_j	Inference job submitted to the queue, $\mathbb{Q}$
$D_{a\_b_j}$	Deadline of job a_b_j
G_k	A partition-enabled local GPU
gi	Valid MIG partition on GPU G
C_{G_k}	Predefined valid configurations per GPU
$JCT_{a\_b_j,g_i}$	Predicted JCT of a_b_j on g_i
$RT_{a\_b_j,g_i}$	Relative performance of a_b_j on a g_i
m_{G_k}	Weight representing the proportion of GPU which is currently being utilized
T_{G_k}	Total system throughput achieved on a local GPU
bT_{G_k}	Highest expected throughput on a local GPU
SP_{G_k}	Set of feasible placements $(a\_b_j, g_i), ..., (a\_b_m, g_n)$ on GPU G_k
W	Level of concurrency for GPU sharing, the maximum of which is $\|G\|$
B	A set of inference jobs evaluated for placement on a GPU as a batch
R_{avail}	Available GPU resources for allocation
R_{req}	Required resources for executing inference jobs in Batch, B
$E_{a\_b_j}$	Time spent executing a_b_j
C_r	Cost of reconfiguring a local GPU
C_w	Cost of waiting for active jobs to complete
β	Threshold for reconfiguration
x_{G_k}, y_{i,a_b_j}	Binary variables to indicate architectural and execution constraints
z_{g_i,g_i}, s_{g_i}	Binary variables to indicate merge or split respectively

Architectural constraints

GPU resource sharing varies by architecture and vendor. AMD uses flexible CU allocation and MxGPU virtualization, while NVIDIA's MIG provides hardware-enforced partitioning for predictable performance. In NVIDIA GPU architectures (A30, A100, H100, etc.) MIG partitions a GPU, G_k, into predefined configurations, C_{G_k}, each comprising of a unique set of MIG instances, g_i per G_k. Only one configuration can be active per GPU at a time, indicated by a binary variable $x_{G_k} \in \{0, 1\}$:

$$\sum_{i \in C_{G_k}} x_{G_k} = 1, \quad \forall G_k \tag{6}$$

Execution constraints

Each inference job, a_b_j, must be placed on a valid GPU partition, g_i, within the feasible placement set $SP_{G_k} = ((a\_b_j, g_i), \ldots, (a\_b_m, g_n))$. Jobs and partitions must adhere to the following rules: ***First,*** a single job cannot be assigned to multiple partitions simultaneously. ***Second*** multiple jobs cannot execute on the same partition concurrently. ***Third,*** after a job a_b_j is completed, its assigned partition becomes available for reuse. The binary variable $y_{i,a\_b_j} \in 0, 1$ indicates whether a job a_b_j can be placed on a partition g_i:

$$\text{If } y_{i,a\_b_j} == 1, \implies \text{ job } a\_b_j \text{ can execute}$$
$$\sum_{i \in g_i} y_{i,a\_b_j} \leq 1, \quad \forall a\_b_j \tag{7}$$

$$y_{i,a\_b_j} + y_{i,a_{j+1}} \leq 1, \quad \forall i, a\_b_j \neq a_{j+1}$$
$$\text{If job } a\_b_j \text{ completes} \Rightarrow y_{i,a\_b_j} = 0 \tag{8}$$

Reconfiguration Last Placement, RLP

Dynamic GPU partitioning minimizes makespan while adapting to evolving resource demands. When new jobs arrive with different resource requests yet logical resources cannot meet resource demands due to physical partitioning constraints, reconfiguration is required to harvest resource fragments and ensure jobs are placed efficiently to maximize system throughput and minimize makespan. This often happens when a job is completed and there are other jobs waiting in the queue. During reconfiguration, all active jobs must be preempted, introducing overhead costs (C_r).

Dyna-P first evaluates whether merging or splitting available resources is feasible and beneficial. For instance, while merging 2g and 1g is infeasible on an NVIDIA A30 GPU, it is supported on A100 and H100 GPUs. Additionally, only partitions adjacent each other can be merged. Binary variables $z_{g_i,g_i} \in 0, 1$ and $s_{g_i} \in 0, 1$ indicate whether a partition can be merged or split respectively:

$$\text{If } z_{g_i,g_i} == 1, \implies \text{Instances can be merged} \tag{9}$$

$$\text{If } s_{g_i} == 1, \implies \text{Instance can be split} \tag{10}$$

If reconfiguration is remain necessary due to infeasible merge or split operations, Dyna-P evaluates whether reconfiguration is worthwhile despite the potential throughput gain from utilizing idle partitions. This requires considering the cost of stopping active jobs and reconfiguring (C_r) the full GPU, and the cost of waiting (C_w) for all active jobs to complete in order to run the next eligible job in the queue despite the logical availability of resources. The scheduler must balance the trade-off between reconfiguring the GPU (C_r) to improve throughput gains and waiting until other partitions become physically available (C_w).

From Equation 11, the reconfiguration cost C_r includes: α, a fixed overhead incurred during the reconfiguration process, $E_{a\_b_j}$, the time already spent by active jobs in batch $B(a\_b_j)$, during execution and a penalty pen_j, if QoS is not met. β is the maximum acceptable cost difference for reconfiguration. In order for reconfiguration to take place, Dyna-P first ensures that all active jobs are expected to meet their deadline despite preemption.

$$\begin{aligned} &C_r - C_w \leq \beta \\ &\forall i, \quad time^{current}_{a\_b_j,g_i} + JCT_{a\_b_j,g_i} \leq D_{a\_b_j} \end{aligned}$$

where

$$\begin{aligned} E_{a\_b_j} &= time^{current}_{a\_b_j,g_i} - time^{start}_{a\_b_j,g_i} \\ C_w &= JCT_{a\_b_j,g_i} - \max\left(E_{a\_b_j} + pen_j\right) \\ C_r &= \alpha + \max_{a\_b_j \in B_{\text{stopped}}} (E_{a\_b_j}), \quad \forall B(a\_b_j) \end{aligned} \tag{11}$$

The Reconfiguration Last Placement policy ensures that reconfiguration is performed only as a last resort, prioritizing resource assignments to existing partitions whenever possible to minimize the frequency of costly reconfigurations.

3.2.1 Dyna-P Scheduler

Dyna-P schedules inference jobs using a branch-and-bound (BnB) algorithm to plan placements on GPUs and leverages the *merge* and *split* features of NVIDIA MIG to minimize fragmentation on GPUs.

Scheduling begins when users submit jobs to the queue, $Q = (a\_b_j, ..., a\_b_m)$. Each user provides QoS requirements (e.g., deadline $D_{a\_b_j}$) for execution. The profiler profiles applications off-line on a First-Come-First-Served (FCFS) basis and stores

9

each profile, $P(p_{a\_b_j}, ..., p_{a\_b_m})$, in a repository. Performance counters collected during off-line profiling with DCGM include *GRACT, DRAMA, TENSOR, MEMORY USAGE, and JCT*. These counters provide a detailed understanding of the behavior of each job in various GPU configurations, and are used by the partition predictor (`PREDPART`) and performance estimator (`PERF`) during scheduling.

Algorithm 2 shows the joint allocation, placement and scheduling of inference jobs on multiple GPUs in a single node to meet the QoS requirements of the user while improving resource utilization and throughput. While the queue Q is not empty ($|Q| > 0$), the partition predictor (`PREDPART`) determines the required resources for each inference job. The performance estimator (`PERF`) predicts the JCT of the job based on profiled metrics and updates the information of each job (lines 2-7).

The scheduler selects the GPU with the minimum Fragmentation Measure ($min_{FM} > 0$, Equation 12) in Line 8 to ensure effective utilization. By so doing, smaller jobs are prioritized for execution, reducing the Head-of-Line blocking that can occur otherwise.

$$min_{FM} = 1 - \frac{\sum_{i=1}^{m} \text{Alloc\_partitions}_i}{\text{Total\_partitions}} \quad (12)$$

For each GPU, Dyna-P calls Algorithm 3 in Line 13, to find the batch of jobs $BestB$, that best maximizes throughput. The window size or the number of jobs considered for scheduling at a time, W, is dynamically tuned based on available resources (R_{avail}) until the maximum concurrency for the GPU architecture, $|G|$, is reached.

Algorithm 2 Dyna-P Scheduler

Input: $Q = (a\_b_j, ..., a_m)$, $P(p_{a\_b_j}, ..., p_{a_m})$, W, $G = (G_1, \ldots, G_k)$

1: **function** SCHEDULER(P, Q, W, G_k)
2: **while** $|Q| \neq 0$ **do**
3: **for** a_b_j **in** Q **do**
4: $g_i \leftarrow$ PREDPART($p_{a\_b_j}$)
5: $JCTa\_b_j, g_i \leftarrow$ PERF($p_{a\_b_j}$)
6: $Q(a\_b_j) \leftarrow Update(g_i, JCTa\_b_j, g_i)$
7: **end for**
8: $G \leftarrow$ SORT(G, min_{FM})
9: ▷ Minimize fragmentation
10: **for** G_k **in** G **do**
11: $W \leftarrow$ SETW($Q, G_k(R_{avail})$)
12: ▷ Set concurrency
13: $BestB \leftarrow$ BNB(Q, W)
14: **if** $BestB == \emptyset$ **then**
15: continue to next G_k
16: **end if**
17: ▷ MIG controller
18: **if** $SP_{G_k} \neq SP_{BestB}$ **then**
19: **if** $z_{g_i,g_i} == 1$ **then**
20: $Merge(SP_{G_k}, SP_{BestB})$
21: **else**
22: **if** $s_{g_i} == 1$ **then**
23: $Split(SP_{G_k}, SP_{BestB})$
24: **else**
25: $C_r \leftarrow C_r(SP_{G_k}, SP_{BestB})$
26: $C_w \leftarrow C_w(SP_{BestB}, SP_{G_k})$
27: **if** $C_r - C_w <= \beta$ **then**
28: continue to next G_k
29: **else**
30: $Reconfig(SP_{G_k}, SP_{BestB})$
31: **end if**
32: **end if**
33: **end if**
34: **end if**
35: **for** a_b_j **in** $BestB$ **do**
36: ASSIGN(a_b_j, g_i)
37: **end for**
38: **end for**
39: **end while**
40: **end function**

Algorithm 3 Job Placement with BnB

Input: Q, W
Initialize: $bT_{G_k} = 0, B = \emptyset, T_{G_k}, BestB = \emptyset$

1: **function** BNB(Q, W)
2: **for** $B \subseteq Q$, where $|B| \leq W$ **do**
3: **if** $R_{req}(\text{B}) \leq R_{avail}(G_k)$ **then**
4: $T_{G_k} \leftarrow m_{G_k} \cdot \sum_{i=1}^{i<W} RT(a\_b_j, g_i)$
5: **if** $T_{G_k} > bT_{G_k}$ **then**
6: $bT_{G_k} \leftarrow T_{G_k}$
7: $BestB \leftarrow B$
8: **end if**
9: **else**
10: Prune subset B
11: ▷ Resource constraints
12: **end if**
13: **end for**
14: **if** $bT_{G_k} == 0$ **then**
15: **return** $\emptyset$ ▷ Log: Invalid Batch
16: **end if**
17: **return** $BestB$
18: **end function**

Output: $BestB$

10

Algorithm 3 ensures that both architectural and execution constraints are met while respecting user-defined QoS. After determining the set of applications with the highest throughput, $BestB$, Dyna-P executes the jobs using the `ASSIGN` function after checking whether placement based on the current partition SP_{G_k} meets the placement demands of $BestB$, SP_{BestB}. This prioritizes the Reconfiguration Last Placement policy and minimizes unnecessary reconfigurations. After reconfigurations, jobs are assigned to partitions that maximize throughput while reducing fragmentation and overhead (Lines 18-36).

Worst-case Analysis

Algorithm 2 runs until all jobs(a_b_j) in the queue are scheduled. Dyna-P's BPP (Algorithm 1) has a time complexity of $O(|Q| \cdot |G|)$, where $|Q|$ is the number of jobs in the queue and $|G|$ is the number of partitions possible on the GPU architecture. Sorting the GPUs based on the min_{FM} takes $O(|G|log|G|)$. The branch-and-bound algorithm efficiently determines suitable allocations by exploring possible combinations of concurrent executions to maximize system throughput while reducing the exhaustive search space [38] from $|Q!|$ to $O(2^{|W|})$. The worst-case time complexity of Dyna-P's scheduler is therefore $O(W \cdot (|G| \log |G| + 2^{|W|})$, where W is the number of jobs from queue Q evaluated, $|G|$ is the number of partitions, and $2^{|W|}$ is the worst-case for Algorithm 3

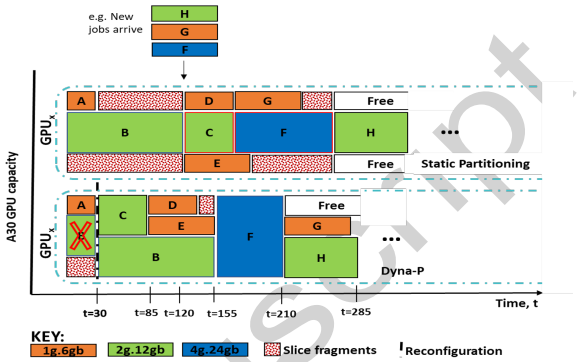
3.2.2 Implementation

Dyna-P, implemented with Bash and Python scripts, integrates all components (Figure 5) and scheduling across multi-GPUs in a node. Specifically, Dyna-P's BPP is implemented in Python to match job profiles with the most suitable GPU partitions. The Performance Estimator uses XGBoost regression[36] to estimate job completion times based on performance counters. The branch-and-bound algorithm, implemented in Python, identifies the set of Jobs $BestB$ which maximize throughput. With bash scripts, Dyna-P evaluates the system for reconfigurations and autonomously manages partition placements and UUIDs through NVIDIA's MIG APIs[39].

Figure 7 illustrates two scheduling scenarios: (i) legacy scheduling using static partitioning with FCFS-based execution, and (ii) Dyna-P.

Job	Arrival time	Execution time	Deadline	Assigned resource	Status
A	0	30	100	1g	Completed
B	0	120	200	2g	Completed
C	5	50	85	2g	Completed
D	60	50	130	1g	Completed
E	60	70	160	1g	Completed
F	120	70	250	1g	waiting
G	120	110 / 50	220	2g / 4g	Running
H	120	80	300	2g	waiting

(a) Job arrival schedule



(b) Job scheduling and execution with time

Fig. 7: Scheduling example with static partitioning and Dyna-P

At time, $t = 30$ for instance, although resources are logically available, they remain physically unavailable due to improper placement, leading to under-utilization. Job C has to wait in the queue, reducing the overall throughput of the GPU. With Dyna-P, resources are dynamically evaluated for reconfiguration and reallocated, enabling job C, to execute while considering active job deadlines. This adaptive reconfiguration utilizes fragmented slices, ensuring jobs like C and F complete on time and reduces overall Makespan(t=285). In contrast, static partitioning suffers from head-of-line (HOL) blocking from job B, preventing timely execution of jobs C and F, leading to missed deadlines and longer Makespan.

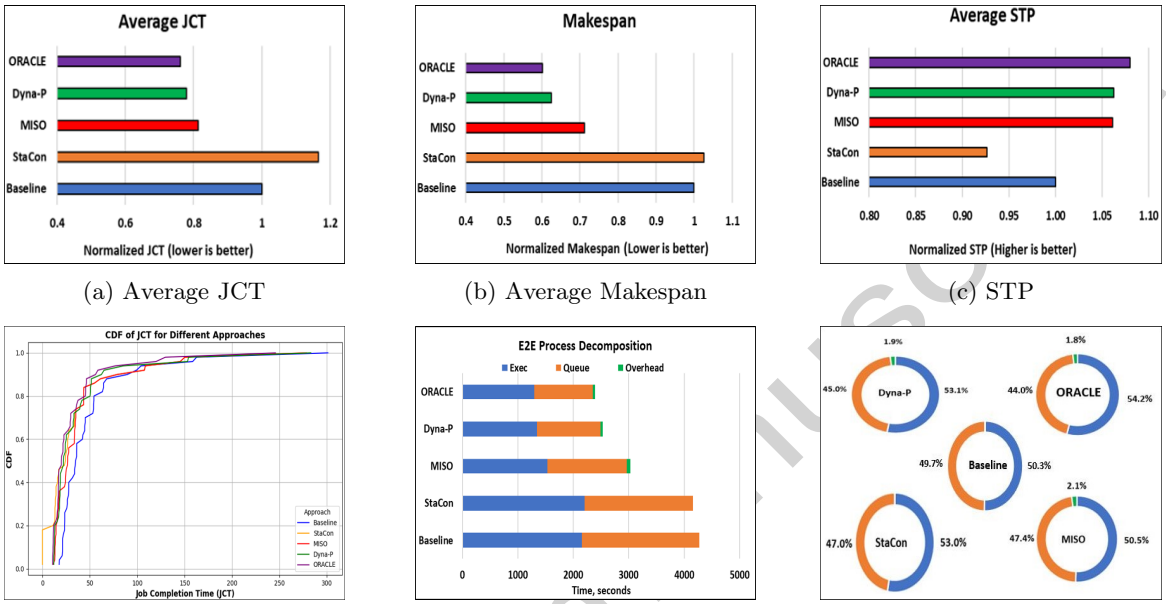
4 Evaluation

Dyna-P is evaluated for efficiency, scalability, and sensitivity to workload and scheduling characteristics using the system configuration described in Table 2. The experiments are carried out on a system equipped with two MIG-enabled

11

Table 2: System Configuration and Inference Job Description

Component	Specification	Model	Params	Size	Description
GPU Device(s)	A30 x 2	DistilGPT	82 M	Small	Text generation
Memory & Bandwidth	24GB & 933.1 GB/s	Codegen	350 M	Medium	Code generation
Thermal Design Power	165 W	Flan-T5-Large	770 M	Medium	Text-to-text
CUDA Ver. & Drivers	12.2 & 535.171.04	Codellama	1 B	Large	Code generation
DCGM Version	3.1.3	Phi-1	1.3 B	Large	General purpose



(a) Average JCT (b) Average Makespan (c) STP

(d) CDF of JCTs (e) E2E Process Decomposition (f) Process decomposition (%)

Fig. 8: Performance and utilization comparison between different approaches.

NVIDIA A30 GPUs to support dynamic partitioning. NVIDIA DCGM is used to collect profile metrics, and CUDA version 12.2 ensures compatibility with the latest GPU features.

A total of 50 Small Language Model (SLM) inference jobs with variants based on batch sizes ranging from 1 to 64 are submitted to the system. The evaluation focuses on three major metrics: *Job Completion Time (JCT)* which is the time taken to complete individual inference jobs, *Makespan* which is the total time required to execute all submitted jobs and *System Throughput (STP)* [32][37] defined as the weighted sum of throughput per partition, normalized to a full GPU.

Dyna-P is compared with alternative sharing approaches: Baseline, StaCon, MISO[32] and ORACLE. For StaCon, a pre-set MIG configuration (2g.12gb, 1g.6gb, 1g.6gb) is used without workload characterization. This configuration is based on empirical studies of job sets.

Baseline represents executions on a full GPU without sharing resources and serves as a performance baseline for maximum capacity. MISO is a state-of-the-art scheduler adapted to run on NVIDIA A30 GPUs. MISO profiles jobs during initial execution to predict resource allocations for subsequent runs. However, for fair comparisons, all jobs executed using MISO are pre-profiled. ORACLE, like Dyna-P, implements a placement-aware scheme however, it has prior knowledge of the partition sizes, job completion times and the job arrivals and thus does not incur costs due to poor predictions. Dyna-P is thus not expected to outperform ORACLE.

12

4.1 Efficiency Analysis

This section evaluates Dyna-P's ability to improve system throughput (STP) and makespan.

JCT, Makespan and STP

Figure 8 compares Dyna-P's Job Completion Time (JCT), Makespan and System Throughput (STP) with the alternative approaches. From Figure 8a, Dyna-P improves JCT by 33.18% relative to StaCon and offers comparable JCT to MISO for most jobs. In Figure 8b, Dyna-P reduces makespan by 39.03% relative to StaCon and 12.14% relative to MISO by using NVIDIA MIG's merge and split for dynamic spatial sharing. Dyna-P achieves 14.7% higher STP compared to StaCon due to its dynamic resource allocation based on workload characteristics as shown in Figure 8c. ORACLE with prior knowledge experiences slight improvements in Makespan compared to Dyna-P.

Figure 8d illustrates the individual job performance under each approach. Noticeably, StaCon exhibits job failures at JCT=0, due to out-of-memory (OOM) issues, emphasizing the importance of workload characterization. Dyna-P, ORACLE and MISO mitigate these failures by profiling workloads and allocating resources accordingly. In all, Dyna-P tends to achieve slightly lower JCTs for shorter running jobs whilst MISO performs better with longer running jobs. ORACLE shows the same trend as Dyna-P since the gains in ORACLE are mainly observed during placements.

The end-to-end (E2E) execution process is further decomposed to show the activities involved in running all 50 inference jobs using the listed approaches. As shown in Figure 8e and 8f, Baseline experiences the longest queuing time as jobs execute sequentially, delaying new arrivals. Dyna-P and ORACLE, using the RLP policy, reduce queuing times by balancing job restarts to increase concurrent executions while managing waiting times subject to user deadlines. Given that merge and split operations are negligible ($\sim \frac{1}{8}$th secs), the overhead stems from reconfigurations. In contrast, MISO incurs overheads due to reconfiguration on job arrival and check-pointing.

Utilization and Eco-friendliness

The average resource utilization (SM, memory) for each approach is compared in Figure 9. Dyna-P achieves an average of 99.8% SM and 32.5% memory utilization due to efficient concurrent executions. ORACLE and MISO shows similar SM utilization (99.8% and 97.05% respectively) but slightly lower memory utilization. Figure 9 also evaluates peak power utilization of each approach.

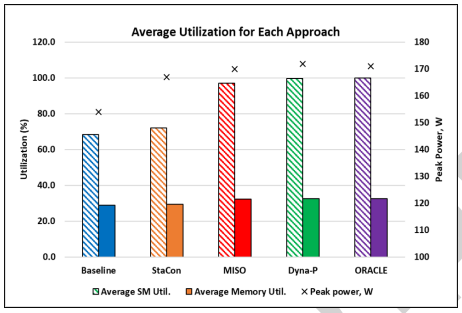


Fig. 9: Utilization Comparison

Both Dyna-P and MISO exhibit peak power usage slightly exceeding the GPU's Maximum Thermal Design Power (165W), increasing the risk of thermal throttling during peak periods. In cluster settings, approaches such as power capping and Dynamic Voltage and Frequency Scaling (DVFS) have been proposed to mitigate thermal throttling in modern GPUs [31].

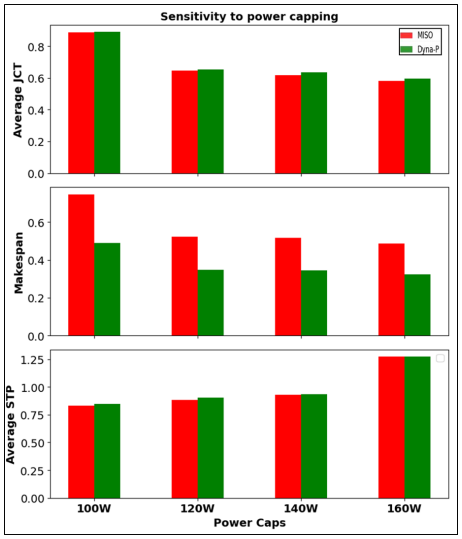


Fig. 10: Power capping for MISO and Dyna-P

Figure 10 explores **NVIDIA's power capping** to mitigate these risks, showing the effects on JCT, Makespan and System Throughput. Under

13

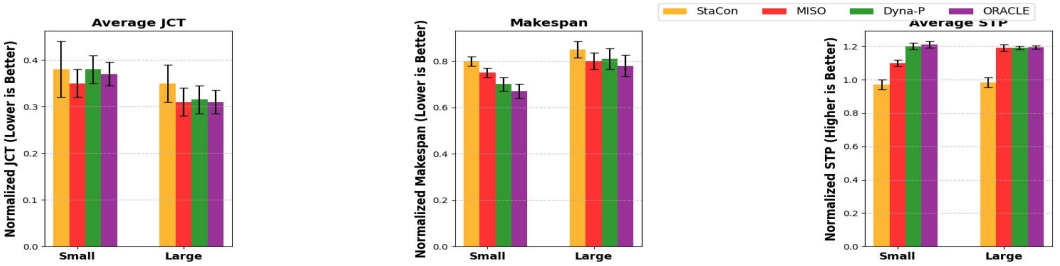


Fig. 11: Workload awareness: Performance comparison for workloads with different batch sizes

capped power limits (100W–160W), STP drops by 33.44% for Dyna-P and 34.80% for MISO. Despite the reduction, Dyna-P's batch-aware partition prediction and ability to harvest free resources result in a lower makespan (50.3% reduction) compared to MISO (53.89%). These findings highlight Dyna-P's eco-friendliness, a critical requirement to reduce thermal throttling and carbon emissions in GPU clusters.

4.2 Workload-awareness Analysis

In Figure 11, the impact of batch sizes on JCT, Makespan, and STP is evaluated using two workload types with 10 inference jobs each on a single A30 GPU: *(a)* small batch sizes(1-8) and *(b)* large batch sizes (32-64) respectively.

Dyna-P achieves higher concurrency by assigning smaller partitions with minimum latency and throughput trade-off to jobs, resulting in better makespan and STP compared to MISO. However, this concurrency slightly increases average JCT due to fewer resources per job.

Dyna-P, ORACLE and MISO outperform StaCon, which suffers from inflexible resource allocations. Dyna-P's dynamic partitioning ensures comparable STP and makespan to MISO, with better resource utilization. In general, Dyna-P demonstrates workload awareness, dynamically adapting resource allocations to maximize throughput and minimize delays.

Also, successful reconfigurations in Dyna-P are affected by the prediction of job completion times using the XGBoost regressor thus, the effect of errors in prediction is evaluated in this section. An error margin of 15% is generated and added to the predicted job completion times, before resource allocations. (Figure 12) shows the comparison of how these errors affect the makespan using each approach as well as the reconfiguration count.

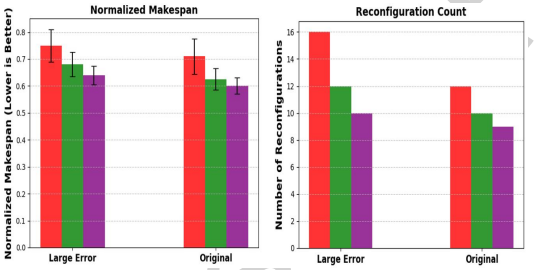


Fig. 12: Sensitivity to Error

It is observed that with the error margin added, Dyna-P experiences 2 additional reconfigurations and 8.09% longer makespan, relative to the original execution. While this is undesirable, Dyna-P has a better makespan by 13.6% than MISO, despite the errors. It also has fewer number of reconfigurations in both cases.

4.3 Scalability Analysis

Dyna-P's scalability is evaluated using two experiments: the first is conducted on a single NVIDIA A30 GPU and the second is a simulation to evaluate the effectiveness of Dyna-P in harvesting GPU resources for the deployment of 5000 jobs on a 20-node GPU cluster where each node has 8 x A100-40GB GPUs. These experiments assess the system's ability to handle an increasing number of jobs and adapt to varying job arrival rates.

Concurrent Jobs

This experiment investigates how each approach performs as the number of concurrently scheduled jobs increases for a single GPU. Figure 13 shows that Dyna-P demonstrates scalability with a gradual increase in Makespan as the number of jobs rises. Dyna-P, ORACLE and MISO exhibit a sharp increase in STP as more jobs are added,

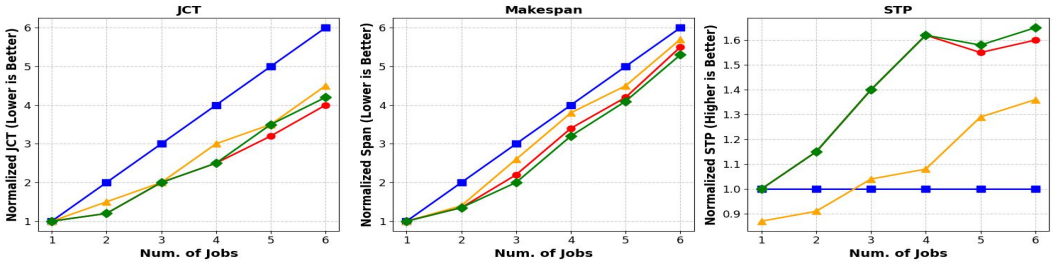
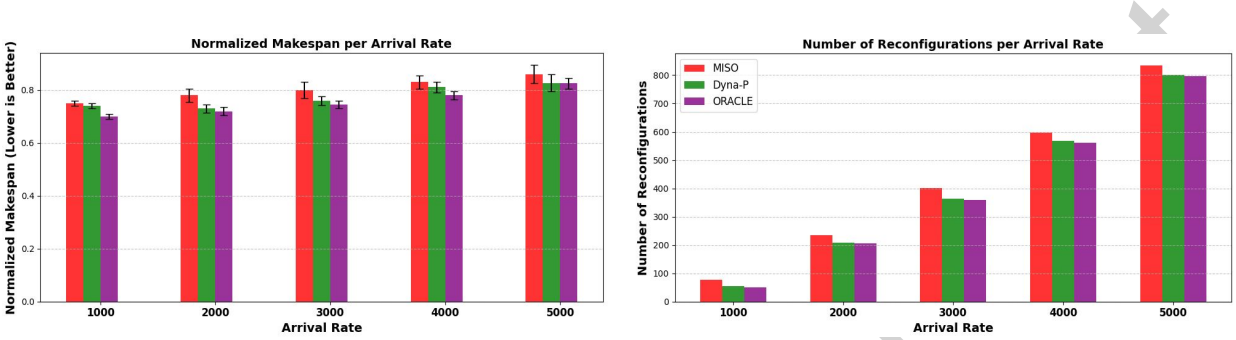


Fig. 13: Performance comparison for increased concurrency on single GPU



(a) Makespan per arrival rate

(b) Number of reconfigurations

Fig. 14: Performance comparison for Large-scale workloads

indicating their ability to handle concurrent workloads effectively. As the number of jobs exceeds four, MISO tends to allocate larger resources over multiple execution rounds, leading to a reduction in STP. In contrast, Dyna-P and ORACLE assign smaller resource partitions for concurrent jobs, maintaining higher STP.

Simulation with Different Arrival Rates

GPU sharing is known to improve resource allocation rates whilst reducing the number of GPUs allocated to jobs. This ultimately reduces the long task wait times. Alibaba workload trace is used to generate a 5000 job-mix for this experiment. Batch sizes from 1 to 64 as in preliminary experiments are assigned to the jobs according to a uniform distribution. For these experiments, a count is kept of the number of scheduling decisions made using each approach as the workloads are increased by 1000 workloads at a time. With the maximum concurrency on the A100-40GB being 7, the simulation environment is able to accommodate up to 1120 concurrent jobs at a time.

From Figure 14, Dyna-P is able to efficiently allocate the minimal necessary GPU resources to jobs using its BPP and thus improves Makespan whilst ensuring user's QoS. It is also able to adapt to the partition profiles of the A100 GPU showing its effectiveness on different GPU architectures. As the arrival rate increases, the makespan for Dyna-P, Oracle, and MISO increase by 11.76%, 17.86%, and 14.67% respectively. Dyna-P and ORACLE, adapting to increased workloads, are seen to harvest more resource fragments and thus experience fewer reconfigurations (748 and 746 respectively). MISO, on the other hand, increases the number of reconfigurations to 756 GPUs, unable to make use of the resource fragments available across GPUs and leading to scheduling delays similar to the scenario in Figure 1.

5 Related Work

Extensive research has explored GPU sharing techniques, particularly focusing on fine-grained and coarse-grained spatial sharing methods to maximize resource utilization. Table 3 provides a comparative overview of related works and highlights the distinct contributions of Dyna-P.

15

Table 3: Comparison of Related Works and Dyna-P

Research Work	Sharing Type	FR	P	MG	DR
BoF[12]	MIG	✓	×	✓	×
MISO[32]	MIG/ MPS	×	×	✓	✓
Gpulet[38]	MPS	×	×	×	×
GSLICE[40]	MIG	×	×	✓	×
Orion[41]	CUDA streams	×	×	×	×
Dyna-P	MIG	✓	✓	✓	✓

FR: Fragmentation Reduction, *P*: intra-GPU Placement, *MG*: Multi-GPU, *DR*: Dynamic Reconfiguration

GSLICE [40] identifies knee-points of diminishing returns for resource allocation and leverages adaptive batching for inference workloads to ensure SLO requirements. However, the MPS-based sharing employed is limited to single GPU environments.

Tan et al. [11] framed DNN serving on MIG-enabled GPUs as a reconfigurable machine scheduling problem. Their genetic algorithm improves GPU partition assignments to maximize throughput and minimize latency relative to SLOs. However, their approach allocates one job per partition without maximizing co-sharing within MIG partitions.

Li et al. [32] proposed a method for predicting isolated GPU partition sizes for a job using MPS. While this improves isolation and performance, their approach focuses on accurate partitioning rather than adapting to diverse workloads. Additionally, MISO does not maximize the resource utilization of per GPU as it focuses on load balancing rather than minimizing fragmentation, limiting its effectiveness in multi-GPU scenarios.

Weng et al. in their paper, Beware of Fragmentation (BoF) [12], leverage a CUDA Runtime API interception approach deployed in a production cluster that runs a mixture of training and inference tasks and increases resource allocation rates by minimizing fragmentation. While they discuss the problem of fragmentation in GPU clusters, they do not consider the inherent intra-GPU placement constraints of modern GPUs.

GPUpool [37] demonstrated fine-grained sharing through simulations by introducing a programmable kernel launch parameter to control concurrency. This approach improves QoS for large batches and minimizes CUDA and MPS limitations through interference modeling however, it does not guarantee isolation.

Orion [41] explored CUDA streams to co-locate inference workloads on shared GPUs demonstrating workload-awareness during resource sharing. Although this improves utilization, it does not support multi-GPU scenarios or incorporate workload diversity.

Dyna-P addresses the gaps in existing approaches by predicting suitable GPU partitions based on workload-specific characteristics, ensuring efficient resource allocation and minimizing under-utilization. Leveraging NVIDIA MIG's merge and split functionalities to dynamically adapt to evolving workloads, Dyna-P reduces fragmentation and improves concurrency in multi-GPU environments while maintaining high throughput and QoS guarantees.

6 Discussions

Recent advancements in green computing, resource-efficient application development, and GPU partitioning schemes highlight the growing need for dynamic resource sharing. Modern GPU architectures such as NVIDIA's Ampere, Hopper and Blackwell, as well as AMD's MI300[42, 43] feature similar GPU partitioning schemes, making Dyna-P adaptable across different platforms. Dyna-P enables efficient workload-to-partition mapping, improving resource allocation to maximize throughput and addressing fragmentation issues arising from inefficient placements in shared GPU environments and can easily be integrated with orchestrators like Kubernetes. This section examines Dyna-P's operation with heterogeneous workloads and discusses its architectural limitations.

Inference in EdgeAI

GPUs are increasingly driving EdgeAI innovation, serving as high-end edge servers that enhance performance, reduce latency, and improve scalability for AI/ML, vision, security, and other edge computing applications. As edge GPU platforms

16

become more capable, lightweight models are frequently deployed at the edge, leading to dynamic inference request patterns that require adaptive resource scaling to prevent fragmentation and under-utilization.

Dyna-P's directed bipartite graph approach allows for seamless addition and removal of partition profiles and model variants based on key characteristics such as parameter count, batch sizes, and utilization patterns. This adaptability ensures efficient scheduling of inference jobs, improving model and partition selections for maximum throughput, particularly as small language models (SLMs) gain popularity and evolve.

Mixed Inference and Training Workloads

In environments such as autonomous vehicles, Neural Network must continuously adapt to dynamic conditions as vehicles move. This necessitates simultaneous GPU resource provisioning for both inference and training tasks, requiring effective resource sharing to minimize wait times and maximize utilization.

To achieve this, Dyna-P's Batch-aware Partition Predictor (BPP) determines the number of requests to batch and the most suitable resource-to-batch size ratio. By leveraging profiled hardware performance counters, Dyna-P estimates model behavior and job completion times, allowing for simultaneous scheduling of inference and training requests. This minimizes reconfigurations, ensuring efficient dynamic resource allocation without compromising performance.

High Performance Computing Workloads

Research on HPC resource allocation [9, 8, 31, 37, 41] has shown that certain scientific benchmark applications (e.g. SCAN, LavaMD, Heartwall, Gaussian) do not fully saturate GPU resources, making them suitable for GPU sharing. Prior work [9, 8] has demonstrated that compute and memory intensities can be leveraged to further optimize GPU resource sharing at a finer granularity. Dyna-P can extend this approach by maximizing resource utilization for tenants running multiple jobs with varying compute and memory demands, while still benefiting from the isolation provided by hardware-level partitioning. This ensures efficient GPU allocation for heterogeneous HPC workloads, improving both job throughput and system efficiency.

Limitations

As discussed, Dyna-P is designed to schedule diverse workloads that require partial GPU resources. Thus, large-scale AI/HPC applications like Large Language Models (LLMs) that require multiple GPUs for training and inference, are beyond the scope of the current implementation.

Also, Dyna-P is unable to migrate active jobs when the GPU requires reconfiguration. This is as a result of the lack of inter-partition communication in modern GPU architectures. This restriction reduces the potential for real-time load balancing and elastic resource allocation for bursty workloads during execution. Enhancing partition communication capabilities could further improve dynamic GPU reconfiguration, allowing for more efficient utilization of fragmented resources in cluster-wide deployments.

7 Conclusion

In this paper, we introduced Dyna-P, a resource allocation, job placement, and scheduling framework designed to improve GPU utilization in multi-tenant environments. By analyzing workload characteristics and using NVIDIA's *merge* and *split*, Dyna-P effectively assigns GPU partitions and co-locates compatible workloads to maximize resource usage while maintaining workload performance.

Our evaluation shows that addressing resource fragmentation enables Dyna-P to improve system throughput and minimize Makespan by harvesting unused resources for other jobs. Its combination of fine-grained and coarse-grained sharing strategies provides a flexible, workload-aware approach to resource management, making it suitable for multi-GPU environments in modern GPU clusters.

Acknowledgments. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No.2021R1A2C1003379).

17

References

[1] Amazon Web Services. *Recommended GPU instances*. Last accessed, December 22, 2024. URL: https://docs.aws.amazon.com/ko_kr/dl-ami/latest/devguide/gpu.html.

[2] IBM Cloud. *IBM Cloud Server's NVIDIA GPU*. Last accessed, December 22, 2024. URL: https://www.ibm.com/kr-ko/cloud/gpu?mhsrc=ibmsearch_a&mhq=GPU.

[3] Google Cloud. *Cloud GPU*. Last accessed, December 22, 2024. URL: https://cloud.google.com/gpu?hl=ko.

[4] Run.ai. *GPU Scheduling*. Last accessed, December 22, 2024. URL: https://www.run.ai/guides/multi-gpu/gpu-scheduling.

[5] Ollama. *Ollama*. Last accessed, December 22, 2024. URL: https://ollama.com/.

[6] Google Cloud. *About GPUs in GKE*. Last accessed, December 22, 2024. URL: https://cloud.google.com/kubernetes-engine/docs/concepts/gpus.

[7] KubeEdge. *KubeEdge: A Kubernetes Native Edge Computing Framework*. Last accessed, December 22, 2024. URL: https://kubeedge.io/.

[8] T. Adufu, J. Ha, and Y. Kim. "Exploring the Diversity of Multiple Job Deployments over GPUs for Efficient Resource Sharing". In: *38th International Conference on Information Networking (ICOIN 2024)*. 2024.

[9] T. Adufu, J. Ha, and Y. Kim. "An Analysis of Efficient GPU Resource Sharing for Concurrent HPC Application Executions". In: *KNOMS Review* 25.1 (Sept. 2022).

[10] Wikipedia Contributors. *Bin-packing problem*. Last accessed December 22, 2024. URL: https://en.wikipedia.org/wiki/Bin_packing_problem.

[11] C. Tan et al. *Serving DNN Models with Multi-Instance GPUs: A Case of the Reconfigurable Machine Scheduling Problem*. 2021. URL: https://doi.org/10.48550/arXiv.2109.11067.

[12] Q. Weng et al. "Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent". In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 2023, pp. 995–1008. DOI: 10.5555/3555555.3569555.

[13] A. Dhakal et al. *Spatial Sharing of GPU for Autotuning DNN models*. arXiv preprint arXiv:2008.03602, 2020. URL: https://arxiv.org/abs/2008.03602.

[14] A. Ferikoglou et al. "Resource Aware GPU Scheduling in Kubernetes Infrastructure". In: *PARMA-DITAM@HiPEAC*. 2021.

[15] M.-C. Chiang and J. Chou. "DynamoML: Dynamic Resource Management Operators for Machine Learning Workloads". In: *CLOSER*. 2021.

[16] G. Yeung et al. "Towards GPU Utilization Prediction for Cloud Deep Learning". In: *USENIX Workshop on Hot Topics in Cloud Computing*. 2020.

[17] NVIDIA. *Multi-Process Service (MPS)*. Last accessed, October 17, 2023. URL: https://docs.nvidia.com/deploy/mps/index.html.

[18] H. Zhao et al. "Tacker: Tensor-CUDA Core Kernel Fusion for Improving the GPU Utilization while Ensuring QoS". In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2022, pp. 800–813. DOI: 10.1109/HPCA53966.2022.00064.

[19] H. Zhao et al. "Exploiting Intra-SM Parallelism in GPUs via Persistent and Elastic Blocks". In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. 2021, pp. 290–298. DOI: 10.1109/ICCD53106.2021.00054.

[20] NVIDIA. *Multi-Instance GPUs*. Last accessed, October 17, 2023. URL: https://docs.nvidia.com/datacenter/tesla-/mig-user-guide/index.html.

[21] Michael Larabel. *The AMD Radeon Graphics Driver Makes Up Roughly 10.5% Of The Linux Kernel*. Phoronix.com, October 2020. URL: https://www.phoronix.com/scan.php?page=news_item&px=Linux-5.9-AMDGPU-Stats.

[22] Zhenyan Lu et al. *Small Language Models: Survey, Measurements, and Insights*. 2024. arXiv: 2409.15790 [cs.CL]. URL: https://arxiv.org/abs/2409.15790.

[23] Chien Van Nguyen et al. *A Survey of Small Language Models*. 2024. arXiv: 2410.20011 [cs.CL]. URL: https://arxiv.org/abs/2410.20011.

[24] Hyung Won Chung et al. *Scaling Instruction-Finetuned Language Models*.

18

2022. DOI: 10.48550/ARXIV.2210.11416. URL: https://arxiv.org/abs/2210.11416.

[25] Suriya Gunasekar et al. "Textbooks Are All You Need". In: *arXiv preprint arXiv:2306.11644* (2023).

[26] Kshiteej Mahajan et al. *Themis: Fair and Efficient GPU Cluster Scheduling*. 2019. arXiv: 1907.01484 [cs.DC]. URL: https://arxiv.org/abs/1907.01484.

[27] Hanyu Zhao et al. "HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 515–532. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/zhao-hanyu.

[28] Zhenhua Han et al. "Scheduling Placement-Sensitive BSP Jobs with Inaccurate Execution Time Estimation". In: July 2020, pp. 1053–1062. DOI: 10.1109/INFOCOM41043.2020.9155445.

[29] Erik Nijkamp et al. *A Conversational Paradigm for Program Synthesis*. 2022. DOI: 10.48550/ARXIV.2210.11416. URL: https://arxiv.org/abs/2210.11416.

[30] Hyung Won Chung et al. *Scaling Instruction-Finetuned Language Models*. 2022. DOI: 10.48550/ARXIV.2210.11416. URL: https://arxiv.org/abs/2210.11416.

[31] E. Arima et al. *Optimizing Hardware Resource Partitioning and Job Allocations on Modern GPUs under Power Caps*. 2023. URL: https://doi.org/10.1145/3547276.3548630.

[32] B. Li et al. "MISO: exploiting multi-instance GPU capability on multi-tenant GPU clusters". In: *Proceedings of the 13th Symposium on Cloud Computing*. 2022.

[33] Noman Bashir et al. "Take it to the limit: Peak prediction-driven resource overcommittment in datacenters". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Association for Computing Machinery, 2021, pp. 556–573. DOI: 10.1145/3447786.3456259.

[34] NVIDIA. *Data Center GPU Manager (DCGM) 3.1*. Last accessed, May 7, 2024. URL: https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/feature-overview.html.

[35] Baolin Li et al. "Clover: Toward Sustainable AI with Carbon-Aware Machine Learning Inference Service". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Nov. 2023, pp. 1–15. DOI: 10.1145/3581784.3607034. URL: http://dx.doi.org/10.1145/3581784.3607034.

[36] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[37] X. S. Tan et al. "GPUPool: A Holistic Approach to Fine-Grained GPU Sharing in the Cloud". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2023. DOI: 10.1145/3559009.3569650.

[38] S. Choi et al. "Multi-model Machine Learning Inference Serving with GPU Spatial Partitioning". In: *PARMA-DITAM@HiPEAC*. 2021.

[39] NVIDIA. *GPU Management and Deployment: Multi Instance GPU Management*. Last accessed, January 20, 2025. URL: https://docs.nvidia.com/deploy/nvml-api/group__nvmlMultiInstanceGPU.html.

[40] A. Dhakal, S. G. Kulkarni, and K. K. Ramakrishnan. *GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform*. Accessed December 22, 2024. URL: https://arxiv.org/abs/2011.03897.

[41] F. Strati, X. Ma, and A. Klimovic. *Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications*. 2024.

[42] AMD. *AMD CDNA 3 Architecture*. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf. Last accessed, March 9, 2025.

[43] AMD. *Deep dive into the MI300 compute and memory partition modes*. https://rocm.blogs.amd.com/software-tools-optimization/compute-memory-modes/README.html. Last accessed, March 9, 2025.

19