dLoRA: Dynamically Orchestrating Requests and Adapters for LoRA LLM Serving

Bingyang Wu, Ruidong Zhu, and Zili Zhang, *School of Computer Science, Peking University;* Peng Sun, *Shanghai AI Lab;* Xuanzhe Liu and Xin Jin, *School of Computer Science, Peking University*

https://www.usenix.org/conference/osdi24/presentation/wu-bingyang

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

dLoRA: Dynamically Orchestrating Requests and Adapters for LoRA LLM Serving

Bingyang Wu[1] *Ruidong Zhu*[1] *Zili Zhang*[1] *Peng Sun*[2] *Xuanzhe Liu*[1] *Xin Jin*[1]
[1]*School of Computer Science, Peking University* [2]*Shanghai AI Lab*

Abstract

Low-rank adaptation (LoRA) is a popular approach to fine-tune pre-trained large language models (LLMs) to specific domains. This paper introduces dLoRA, an inference serving system for LoRA models. dLoRA achieves high serving efficiency by dynamically orchestrating requests and LoRA adapters in terms of two aspects: (i) dynamically *merge* and *unmerge* adapters with the base model; and (ii) dynamically *migrate* requests and adapters between different worker replicas. These capabilities are designed based on two insights. First, despite the allure of batching without merging a LoRA adapter into the base model, it is not always beneficial to unmerge, especially when the types of requests are skewed. Second, the autoregressive nature of LLM requests introduces load imbalance between worker replicas due to varying input and output lengths, even if the input requests are distributed uniformly to the replicas. We design a credit-based batching algorithm to decide when to merge and unmerge, and a request-adapter co-migration algorithm to decide when to migrate. The experimental results show that dLoRA improves the throughput by up to 57.9× and 26.0×, compared to vLLM and HugginFace PEFT, respectively. Compared to the concurrent work S-LoRA, dLoRA achieves up to 1.8× lower average latency.

1 Introduction

Large language models (LLMs) are changing the landscape of modern applications. LLMs such as GPT4 [1] and Llama-2 [2] are pre-trained on a large corpus to achieve outstanding capabilities on generic tasks. These pre-trained LLMs (a.k.a. base LLMs) can be fine-tuned to a specific domain to optimize particular application scenarios, e.g., fine-tuning Llama-2 for better code generation [3]. LLM platforms [4–7] provide fine-tuning APIs and services for developers to fine-tune LLMs and build domain-specific applications. For example, OpenAI provides fine-tuning APIs for fine-tuning GPT-4 and Completions API to access these fine-tuned LLMs [4].

Low-rank adaptation (LoRA) [8, 9] is a popular approach to fine-tuning LLMs. It is a type of parameter-efficient fine-tuning [10] that reduces fine-tuning costs by updating only a small portion of model parameters. LoRA exploits the low dimensionality of parameter updates in fine-tuning and represents them with pairs of two small matrices called LoRA adapters. Fine-tuning a base LLM amounts to training a LoRA adapter for a specific domain while keeping the base LLM unchanged. Compared to the fully fine-tuning GPT-3 175B, LoRA can reduce the number of updated parameters by 10,000× and the GPU consumption by 3× while achieving comparable model quality [8]. At inference time, the LoRA adapter can be merged with the base LLM thus introducing no extra inference overhead.

Serving a set of LoRA model fine-tuned on a base LLM (i.e., LoRA as a service) introduces new challenges to LLM inference serving. Existing LLM inference systems such as Orca [11] and vLLM [12] focus on serving a single model, while in the scenario of LoRA as a service, there are multiple models. Conceivably, one can use Orca or vLLM to serve each LoRA model and adopt an existing model serving orchestrators like SHEPHERD [13] and AlpaServe [14] to manage multiple LoRA models. This simple approach does not consider the characteristics of LoRA model serving and has the following two fundamental problems.

First, serving each LoRA model separately introduces a high memory footprint, and suffers from low GPU utilization as the GPUs cannot be efficiently multiplexed across models. The problem is particularly acute for LLMs as LLMs have large sizes. An alternative approach is to directly use *unmerged* LoRA adapters, i.e., keeping the base LLM unchanged and storing the set of LoRA adapters alongside. When serving different types of requests, it can batch the shared base LLM computation across requests to increase efficiency. This unmerged approach, however, does not work well when the requests are skewed on a particular LoRA adapter, whereas the merged approach can further reduce computational costs to increase efficiency.

Second, LLM tasks have variable input and output lengths, which naturally introduces load imbalance between different worker replicas. Due to the autoregressive pattern of LLMs, simply dispatching requests to different worker replicas uniformly does not work well, as the execution time and GPU memory consumption of requests are diverse [11, 12]. The LoRA as a service scenario further exacerbates the problem,

as LoRA adapter orchestration across worker replicas also needs to be taken into consideration. The dependency between LoRA adapters and requests as well as the GPU memory competition between them make the problem even harder than traditional load balancing problems.

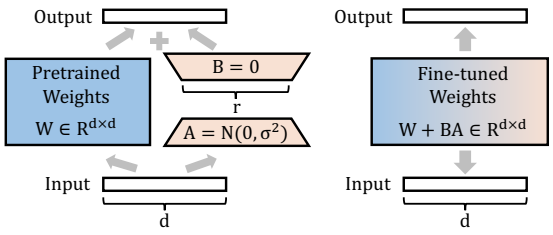
We introduce dLoRA, a new inference serving system for LoRA models to address these two problems. Compared to previous practices in LoRA serving, dLoRA further improves efficiency with two special capabilities. First, dLoRA can dynamically *merge* and *unmerge* LoRA adapters with the base model in each worker replica. Second, dLoRA can dynamically *migrate* LoRA adapters and requests between worker replicas. Exploiting the capabilities of dLoRA efficiently has two technical challenges. The first one is how to decide when to merge and unmerge adapters. The second one is how to decide which adapters and requests to migrate. We propose two techniques to address these two problems.

To address the first problem, we propose a dynamic cross-adapter batching technique. Based on the request arrival pattern and current state, dLoRA dynamically switches between merged and unmerged inference with different batching strategies to reduce the end-to-end latency. To decide an appropriate switching time, dLoRA dynamically adjusts the switching threshold with the thresholds tuning according to the request pattern to reduce switching overhead. dLoRA adopts a credit-based batch generator to generate potentially efficient batching plans without harming the fairness of requests. A final decision is made by taking all factors into account, including execution time, queuing delay, and switching overhead.

To address the second problem, we propose a request-adapter co-migration technique. Except for proactively dispatching requests based on current requests and adapter distribution, it also dynamically migrates requests and replicates adapters to handle unpredictable load imbalance across replicas. We formulate the problem as an integer linear programming (ILP) problem and compute an optimal solution to minimize the load imbalance. To address the high overhead of ILP, we amortize the overhead by reducing the frequency of ILP solving and relaxing the problem to a selective migration problem inspired by selective replication [15–17].

In summary, we make the following contributions.

- We identify the inefficiencies of current LLM serving systems in the LoRA model serving scenario, and articulate the challenges of serving LoRA models.
- At the worker level, we propose a dynamic cross-adapter batching technique to dynamically switch between merged and unmerged modes to reduce the end-to-end latency.
- At the cluster level, we propose a request-adapter co-migration technique to dynamically migrate requests and adapters to balance the load across the cluster.
- We design and implement dLoRA with the preceding two techniques. The evaluation results based on real-world workload traces show that dLoRA achieves up to 57.9× and 26.0× higher throughput than vLLM [12] and HuggingFace PEFT library [10], respectively. By leveraging dynamic orchestration mechanisms, dLoRA also achieves up to 1.8× lower average latency compared to the concurrent work S-LoRA [18].



(a) LoRA fine-tuning. (b) LoRA inference.

Figure 1: LoRA optimization.

2 Background and Motivation

2.1 Parameter-Efficient Large Language Models

Large language models. Large language models (LLMs) try to maximize the next token predictability given the previous tokens. At the inference time, LLMs show an autoregressive pattern. For each request, an LLM iteratively generates tokens based on the prompt (i.e., input tokens) and previous output tokens once a time until it generates an end-of-sentence marker. This autoregressive nature makes LLM inference exhibit two characteristics. The first one is variable inference latency depending on the input and output lengths [12, 19]. The second one is significant GPU memory consumption of intermediate states of requests. To reduce redundant computation, LLMs cache the intermediate states of previous tokens, called key-value (KV) cache, in the GPU memory [20]. Similar to prior work [12], we use intermediate states to refer to the key-value cache. As the size of the KV cache is proportional to the number of input and output tokens, the memory consumption of LLM inference is also variable. The KV cache consumes a large amount of GPU memory and may bound the performance of LLM inference due to the limited GPU memory capacity [12, 19].

Low-rank adaptation. Fine-tuning adapts a pre-trained LLM to a specific domain without training an LLM from scratch. Low-rank adaptation (LoRA) [8, 9, 21] is a popular class of parameter-efficient fine-tuning methods [10], as it can achieve competent performance by only fine-tuning a small number of trainable parameters, called the *adapter*. Furthermore, LoRA does not introduce extra inference latency. Inspired by the phenomenon of low "intrinsic rank" of weight updates, the core of LoRA is to represent each weight update as two rank composition matrices with much smaller ranks. During fine-tuning, LoRA only needs to optimize these two rank composition matrices, while keeping the pre-trained weights frozen. Figure 1 shows an example. For a pre-trained weight W with the shape of $d \times d$, LoRA represents weight updates ΔW as

two smaller matrices A and B with the shape of $r \times d$ and $d \times r$ respectively, where r is much smaller than d. During fine-tuning, as shown in Figure 1(a), LoRA only updates A and B and keeps W frozen, which significantly reduces computation and memory consumption. At the inference time, as shown in Figure 1(b), LoRA can merge the multiplied matrix $B \times A$ (i.e., ΔW) into W to eliminate extra inference overhead. Given the benefits of LoRA, it is widely adopted to enhance the capability of LLMs, such as long sequence [22] and multimodal input [23]. It can be used in all dense layers of LLMs, but it is typically used to adapt attention weights [8].

2.2 Inference Serving Systems

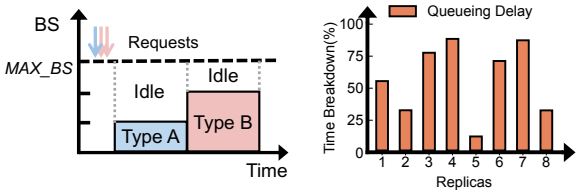
LLM serving systems. Many techniques have been developed to improve the efficiency of LLM inference by leveraging the characteristics of LLMs. Orca [11] proposes iteration-level scheduling to batch requests of different lengths at the granularity of iterations. Completed requests can be removed immediately and newly arrived requests can be inserted into the batch without waiting for the completion of the current batch. The state-of-the-art solution vLLM [12] further introduces an on-demand block-based GPU allocation mechanism called PagedAttention to reduce GPU memory fragmentation caused by variable and unpredictable KV cache, thereby increasing the maximum batch size. However, they only focus on the single LLM serving scenario. When serving multiple LoRA LLMs, they cannot share the common base model and thus cause severe redundant memory consumption.

HuggingFace PEFT [10] is a popular library for parameter-efficient fine-tuning. It can also be used to serve multiple LoRA LLMs shared with the same base LLM. However, it can only serve requests destined to the same adapter once at a time by swapping between different adapters, leading to low efficiency. Besides, it lacks support for cluster-level management for requests and adapters.

Traditional DNN serving systems. Many DNN serving systems can orchestrate multiple DNN models in a cluster, such as SHEPHERD [13] and AlpaServe [14]. However, they also do not support sharing the base model among different models and do not target autoregressive LLMs. PetS [24] can serve multiple parameter-efficient non-autoregressive transformer models in a single server, but it cannot serve autoregressive LLMs and does not consider LoRA. Besides, it does not support cluster-level management for requests and models. In short, existing DNN serving systems also cannot serve multiple LoRA LLMs in the cluster wide efficiently.

2.3 Challenges

To serve a large number of requests, a serving system usually deploys multiple replicas of the same base LLM, with each handling a subset of the requests. Nevertheless, when employing existing systems (such as vLLM and PEFT) to serve LoRA LLMs, we identify two primary challenges.



(a) Challenge within replicas. (b) Challenge across replicas.

Figure 2: Challenges in existing LoRA serving.

GPU underutilization within a replica. In a model replica, a single base model is accompanied by multiple LoRA adapters for different types of requests. The aforementioned PEFT only accommodates batching requests with the same LoRA adapter (i.e., the same type of requests). When serving one type of request, PEFT forces other types of requests to wait until the completion of the current batch. The serving system has to handle low-frequency types of requests one by one, which cause severe GPU underutilization. Figure 2(a) illustrates an example. When three requests arrive simultaneously, even if the max batch size is three, PEFT has to process these three requests separately: one batch for a request destined to the type A adapter and another batch for two requests destined for the type B adapter. As a result, the serving system only utilizes 50% of the total GPU resources and doubles the total latency for the requests destined to the type B adapter. This example indicates that although LoRA does not introduce extra inference latency for a single request, it is still challenging to serve multiple LoRA LLMs efficiently.

Load imbalance across replicas. To manage multiple replicas in a cluster, a serving system usually adopts a global scheduler to dispatch each incoming request to a specific replica. However, there exists load imbalance across replicas from two aspects. First, due to the limited GPU memory, one adapter type may only reside in a subset of replicas. When a burst of requests destined for this adapter type arrives, only a few replicas are utilized, while other replicas are idle. Figure 2(b) shows an example that sending requests based on the Azure trace [25] adopted by a previous DNN serving work [14] to a cluster with eight replicas, where 32 types of adapters are uniformly loaded in the eight replicas. In this case, the burst of requests leads to severe load imbalance across replicas. The difference in queuing delay between replicas can be up to 8.0×. Second, even if the requests are dispatched uniformly across replicas, the variable input and output lengths of requests still lead to load imbalance inevitably. The statistics of ShareGPT [26], the datasets collected from real-world conversations with ChatGPT [27], show that the input and output lengths of requests are highly variable. The longest input length and output length of requests are longer than the average lengths by 636.7× and 163.9× respectively, which implies extremely diverse execution time and GPU memory consumption among requests. The load imbalance caused by variable input and output lengths undermines the system's

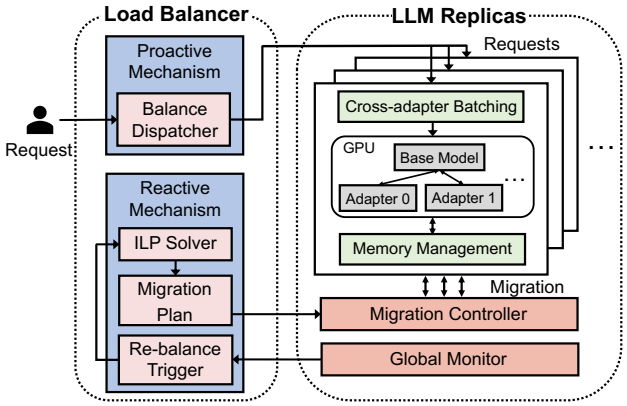


Figure 3: dLoRA architecture.

overall inference efficiency and significantly increases latencies for requests on the overloaded replica.

3 Overview

dLoRA is an inference serving system that serves multiple LoRA models in a cluster. Within a replica, dLoRA employs a novel cross-adapter batching technique to process requests to different LoRA adapters in a single batch and improves the GPU utilization (§4). Across replicas, dLoRA dynamically migrates LoRA adapters and requests in the cluster to achieve better load balancing (§5). Figure 3 shows the overall architecture of dLoRA.

Intra-replica. dLoRA deploys a set of worker replicas in a cluster. Each replica contains a subset of LoRA adapters and one base model on several GPUs.

Dynamic batching. Within a replica, dLoRA uses a local *cross-adapter batching* technique to process requests from the global scheduler. The replica maintains a queue to buffer incoming requests and schedules a batch of requests to the execution engine with dynamic batching to achieve optimal tradeoff between merged and unmerged inference.

Memory management. dLoRA efficiently manages the GPU memory of LoRA adapters and requests within a single replica. The memory manager allocates the GPU memory to the LoRA adapters and the intermediate states of requests. The two different types of memory may race for the limited GPU memory, which harms the serving performance. To mitigate this, the memory manager dynamically adjusts the memory allocation for adapters and requests based on the current GPU memory usage and workload pattern. Besides, the memory manager also swaps the unused adapters and requests to the host memory.

Inter-replica. To manage multiple replicas in the cluster, dLoRA uses a load balancer to solve the aforementioned load imbalance problem. Due to the variable input and output lengths of LLM requests, dLoRA introduces proactive and

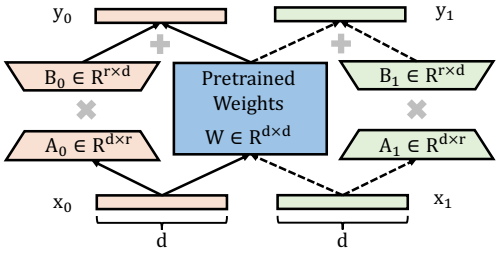


Figure 4: Unmerged inference.

reactive mechanisms to solve the imbalance problem before and after it occurs, respectively.

Proactive mechanism. The proactive mechanism, upon receiving user requests, directs them to a specific replica based on a two-fold dispatching policy. For short-term load dynamics, the mechanism proactively selects the replica with the most available resources to load the corresponding adapter and process the request. For long-term load dynamics, the mechanism proactively loads and replicates adapters for future predictable load spikes.

Reactive mechanism. The proactive mechanism alone is not sufficient since the resource usage (i.e., input and output length) of a request is variable. The load imbalance problem still occurs. To address this issue, dLoRA introduces a reactive mechanism to handle such a situation. Specifically, there is a global monitor that periodically collects the resource usage of each replica. Once the monitor detects a replica with a heavy load, it notifies the re-balance trigger. The reactive mechanism then employs a request-adapter co-migration algorithm to find the optimal migration plan and sends the plan to the cluster's migration controller. The controller then migrates requests' intermediate states and loads LoRA adapters across different replicas to achieve load balancing.

4 Dynamic Batching

4.1 Unmerged Inference

In §2.3, we discuss the limitations of merged inference in terms of significant queuing delay and poor GPU utilization when serving multiple LoRA LLMs. However, the characteristics of LoRA LLMs provides an opportunity.

Unmerged inference. Figure 1(b) shows that the fine-tuned weights and the pre-trained weights are merged to serve a request (i.e., $y = (W + BA)x$), as initially suggested in the LoRA paper [8]. This approach is termed *merged inference*. An alternative approach is *unmerged inference* to process requests with different types in a single batch. Specifically, unmerged inference separates the computation of the pre-trained LLM weights and each LoRA adapter weights. During inference, different types of requests can be batched together to share the same computation with the pre-trained LLM

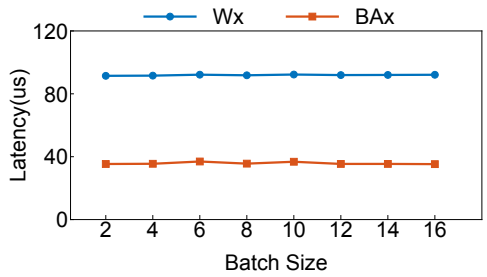


Figure 5: Non-negligible adapter computation overhead.

weights and process different computation with LoRA adapter weights at parallel.

Figure 4 demonstrates unmerged inference. Consider two different requests, x_0 and x_1, with their respective LoRA adapters, $adapter_0$ and $adapter_1$. During inference, the base model inference Wx is batched together as $W \times [x_0, x_1]$. Simultaneously, the LoRA adapter inference BAx is computed separately for each request as $B_0A_0x_0$ and $B_1A_1x_1$ in parallel. Finally, the base model inference and the LoRA adapter inference are aggregated to obtain the final results $y_0 = Wx_0 + B_0A_0x_0$ and $y_1 = Wx_1 + B_1A_1x_1$. Unmerged inference improves the GPU computation efficiency by improving the batch size. However, this solution poses a new challenge.

Extra Computation Overhead. Although unmerged inference accelerates the requests processing with different types, the benefit of unmerged inference technique is not a free lunch. Unmerged inference requires computing three matrix multiplications for each request, i.e., Wx, Ax and BAx, and a matrix addition for result aggregation. As a result, unmerged inference introduces extra computation overhead of two additional matrix multiplications (i.e., separate computation for adapter inference) and one additional matrix addition in each layer.

Figure 5 illustrates the overhead associated with unmerged inference. We conduct an experiment to compare the execution time of the base LLM computation, denoted as Wx (equivalent to original LoRA LLM inference), with the LoRA adapter computation BAx. The experimental setup follows §7. The results, depicted in the figure, reveal that the execution time for the LoRA adapter computation BAx is 38.9% of the baseline LLM computation Wx. This finding suggests that unmerged inference might not always yield performance benefits. On the contrary, performance may degrades when only processing few types of requests.

4.2 Dynamic Batching with Thresholds Tuning

In the preceding discussion, we highlight how unmerged inference incurs additional computational overhead, while merged inference leads to significant queuing delays and low GPU efficiency. A combined approach of these two batching methods can potentially offer a more balanced tradeoff between queuing delay and computational overhead. Yet, determining the optimal batching strategy, within a constrained scheduling time frame, is challenging. First, the decision needs to

Algorithm 1 Dynamic Batching

```
1:  function DYNAMICBATCHING(B_fcfs, R, S, L)
2:      Input: FCFS requests B_fcfs, Request R = {r_1, r_2, ..., r_n}
3:             Replica state S, LoRA adapters L = {l_1, l_2, ..., l_m}
4:      Output: The batch of requests to be executed B_next
5:      // Adaptive switching between different modes
6:      if S.state == unmerge then
7:          R_merge = arg max_{l_i ∈ L} |{r_i ∈ R | r_i.type == l_i}|
8:          if |R_merge|/|B_fcfs| > α_switch then
9:              S.state, S.type = merge, R_merge.type
10:             return B_next = R_merge[: max_bs]
11:         else
12:             return B_next = B_fcfs
13:     else
14:         R_merge = {r_i ∈ R | r_i.type == S.type}|
15:         if |R_merge|/|B_fcfs| < β_switch then
16:             S.state = unmerge
17:             return B_next = B_fcfs
18:         else
19:             return B_next = R_merge[: max_bs]
```

made at the granularity of the iteration while each iteration of LLM inference is typically around tens or hundreds of milliseconds [11, 19]. To avoid the performance degradation, the decision must be made swiftly. Second, the unpredictability of request arrival patterns and request execution time further complicates this decision-making [13, 14, 26]. Last but not least, switching between the two inference methods introduces non-negligible overhead (i.e., an additional matrix multiplication BA and a matrix addition/subtraction between base LLM and adapter). We propose a dynamic batching technique to choose the inference method at runtime.

Algorithm. Algorithm 1 outlines the pseudo-code. Each iteration begins by assessing the replica's state (line 6). In the unmerged state, the algorithm estimates potential performance gains from merged inference. It selects the LoRA adapter with the most requests (line 7) and evaluates its performance against the default first-come-first-serve (FCFS) order (line 8). We use the (FCFS) as the default scheduling choice. Dynamic batching is orthogonal to the underlying scheduling policy; other policies can also be used. If the size ratio exceeds the switching threshold α_{switch}, indicating a performance enhancement, the algorithm switches to merged inference (line 9) and processes the requests from the corresponding adapter type (line 10). If not, it continues with the FCFS batch (lines 11–12). In scenarios of merged inference, the algorithm first batches requests matching the active adapter, i.e., R_{merge} (line 14). If the FCFS batch size ratio to R_{merge} is smaller than the threshold β_{switch} or the active LoRA adapter is not in the set of legal adapters L (line 15), the unmerged state (i.e., processing B_{fcfs}) is deemed advantageous (lines 16–17). Otherwise, the algorithm remains the merged state (lines 19).

Adaptive threshold tuning. The switching threshold α_{switch} and β_{switch} are the key parameters of the algorithm. Figure 6

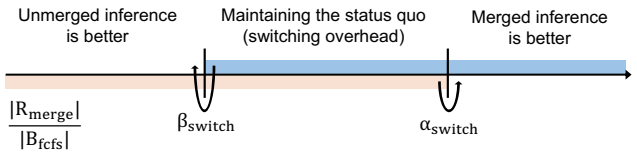


Figure 6: Thresholds demonstration.

illustrates the two thresholds between merged and unmerged inference. The horizontal axis represents the ratio between $|R_{merge}|$ and $|B_{fcfs}|$.

Our first insight is that the switching overhead can be amortized across multiple future iterations. For example, assume that the switching overhead from unmerged to merged inference is 100 ms, and assume that merged inference of one iteration is 50 ms and unmerged inference of one iteration is 100 ms. Let the number of iterations in current processing is three, and the overall execution time is 300 ms if the status is unmerged inference. However, if we switch to merged inference at the beginning, the overall execution time is 250 ms for three iterations, i.e., switching overhead 100ms plus three iterations of 150 ms. In this case, the switching overhead is amortized across the three iterations. Our second insight is that leveraging historical retrospection is possible, despite the unavailability of future knowledge. Continuing with the example, at the beginning of the first iteration, the total number of iterations remains unknown. If requests are finished in one iteration, merging inferences offers no advantage. However, if processing spans two or more iterations, it becomes plausible to anticipate additional future iterations. Under such circumstance, switching to merged inference is still timely.

Based on the two insights, we propose an adaptive threshold tuning algorithm. The algorithm first introduces the break point on iteration granularity, which is marked by events such as replica switching, changes in R_{merge}, or after processing a set number of iterations. Upon reaching a break point, the algorithm tunes the thresholds based on the data collected from the preceding period, stretching from the current to the previous break point. The pseudo-code is outlined in Algorithm 2. In cases where the previous period's replica state is *unmerge*, we focus on tuning α_{switch}. Let the number of the iterations in the previous period be N_I and the merged requests, $R_{merged}[:max_bs]$, in the i_{th} iteration be B_i and B_{fcfs} in the i_{th} iteration be B'_i. The throughput of the merged inference T_{merge} and the throughput of the unmerged inference $T_{unmerge}$ are calculated in lines 4–5. The iteration time can be profiled accurately [19]. A higher T_{merge} leads to a reduction in α_{switch} by a decrement factor γ_{dec}. Otherwise, it is increased by a multiplication factor γ_{mul}. If the replica state in the previous period is *merge*, the algorithm tries to tune β_{switch}. The tuning is similar to the case of α_{switch}, and we omit the details for brevity. This method ensures the adaptability of the two thresholds to varying request characteristics.

Starvation prevention. Although dynamic batching accelerates LoRA LLM serving, it may cause starvation. Specifically,

Algorithm 2 Adaptive Threshold Tuning

1: **Input:** Candidate period N_I, Merged batches $B_1, B_2, ..., B_I$, Switching overhead t_M, Current switching threshold α_{switch}
2: **Output:** New switching threshold α_{switch}
3: **function** ADAPTIVETUNING(N_I, $\{B_i\}$, t_M, α_{switch})
4: $\quad T_{merge} = \frac{\sum_{i=1}^{N_I} |B_i|}{\sum_{i=1}^{N_I} IterationTime(B_i) + t_M}$
5: $\quad T_{unmerge} = \frac{\sum_{i=1}^{N_I} |B'_i|}{\sum_{i=1}^{N_I} IterationTime(B'_i)}$
6: $\quad$ **if** $T_{merge} > T_{unmerge}$ **then**
7: $\quad\quad \alpha_{switch} = \alpha_{switch} - \gamma_{dec}$
8: $\quad$ **else**
9: $\quad\quad \alpha_{switch} = \alpha_{switch} \times \gamma_{mul}$
10: $\quad$ **return** α_{switch}

because dynamic batching prefers processing requests with the most loaded LoRA adapter type, it may starve other types of requests. To address this problem, we use a credit-based mechanism to prevent starvation. The basic idea involves allocating a credit to each LoRA adapter. This credit is then transferred to any preempted adapter. When the credits of certain adapters exceed a threshold, the algorithm prioritizes processing requests with these adapters. Detailed explanations are provided in §A.1.

4.3 LoRA Adapter Offloading

Besides GPU computational resources, GPU memory resources are also heavily used by LLM inference workloads. Sharing a base LLM across LoRA LLMs mitigates the GPU memory footprint, yet memory scarcity persists, especially when storing multiple LoRA adapters. For instance, a single LoRA adapter for Llama-7B requires 56 MB of GPU memory, which is comparable to the intermediate states (i.e., key-value cache) of 7 tokens. As the number of LoRA adapters increases, only a small fraction of GPU memory is available for storing intermediate states of requests. To this end, we employ a swapping mechanism that swaps LoRA adapters and intermediate request states between GPU and host memory. The compact size of LoRA adapter facilitates rapid swapping. Such process is further accelerated by overlapping the swapping with execution using prefetching techniques [19]. Regarding GPU memory distribution, dLoRA adopts a workload-aware allocation algorithm for balancing LoRA adapters and intermediate request states, as detailed in §5.

5 Dynamic Load Balancing

To address the load imbalance problem across replicas for LoRA serving, we combine proactive and reactive mechanisms. Specifically, we follow the existing approach [13, 14] to adopt a proactive mechanism that dispatches requests to replicas. The challenge of LoRA serving as discussed in §2.3 is that the variable input and output lengths of LLM requests cause load imbalance even if the proactive mechanism bal-

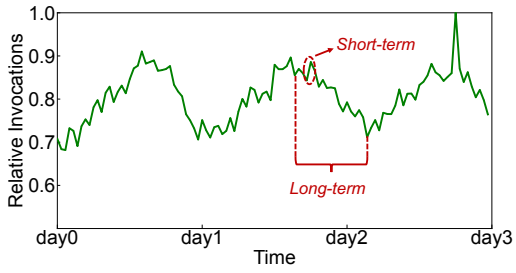


Figure 7: Workload pattern.

ances the load when dispatching requests. Therefore, we design a reactive mechanism that migrates requests and adapters between replicas to reactively address such load imbalance.

5.1 Proactive Mechanism

We first introduce the dispatcher that proactively dispatches requests to replicas and loads LoRA adapters to GPUs according to the workload patterns (i.e., the arrival pattern of requests). Figure 7 shows a production trace used by prior works on inference serving [13, 14]. The workload pattern of this trace can be examined from both long-term and short-term perspectives. In the long term, the pattern exhibits predictability and periodicity, e.g., low load at midnight and high load during daytime. On the other hand, in the short term, the pattern is marked by unpredictability and burstiness. We follow existing work [13, 14] to adopt a proactive mechanism based on these characteristics for LoRA serving.

Guided by the predictable long-term workload, it is possible to preload LoRA adapters to GPU memory to reduce the adapter loading time. During the adapter preload process, we follow the existing work [13] to maximize the minimum burst tolerance of LoRA adapters. The burst tolerance of an adapter in our scenario is defined as the ratio of the peak capacity to the average load of the adapter, where the average load can be obtained from the historical long-term workload. Different from the previous DNN serving scenarios, we find that the peak capacity of a LoRA adapter is dominated by the GPU memory allocation between LoRA adapters and requests' intermediate states. As more LoRA adapter is preloaded into the GPU memory, more replicas can serve this type of request immediately. However, it leaves less GPU memory for requests' intermediate states, which limits the peak serving throughput. Therefore, we define the peak capacity of a LoRA adapter as the number of requests that can be served by the unallocated GPU memory where the corresponding LoRA adapter resides. Based on this metric, the dispatcher of dLoRA greedily preloads the LoRA adapter with the lowest burst tolerance to a replica without this adapter until the minimum burst tolerance decreases to find an optimal placement plan.

Due to the unpredictable short-term pattern of the workload, there also exists a short-term burst of certain LoRA adapters' requests. Such short bursts may cause the burst requests to be dispatched to a small set of replicas with the corresponding LoRA adapters. This leads to severe load imbalance and long queuing delay od requests. On the contrary, if the burst requests are dispatched to other replicas without the corresponding adapter, it incurs additional loading time and consequently harms the serving performance. To address this problem, dLoRA use an adapter-aware dispatch policy with dynamic LoRA loading to consider all these factors. Specifically, dLoRA calculates an estimated pending time for each replica, which includes the time to load the LoRA adapter (if not already loaded) and the estimated queuing time on this replica. Based on this metric, dLoRA dispatches requests to the replica with the shortest estimated pending time to balance the load across replicas. This is especially effective in mitigating issues caused by sudden bursts of requests and consequent queuing delays in heavily loaded replicas.

5.2 Reactive Migration

Although proactive dispatching can mitigate the load imbalance problem, the system still suffers from load imbalance caused by variable input and output lengths of LoRA requests (§2.1). Therefore, even if the workload arrival pattern is stable and entirely predictable, the load imbalance still occurs when some requests in some replicas have longer execution times than others. The GPU memory consumption of requests also suffers from the same load imbalance problem. Some replicas may hold substantial intermediate states of requests even larger than the GPU memory capacity, which causes frequent swapping between GPU memory and host memory. Other replicas may hold a smaller amount of intermediate states of requests. Such load imbalance harms the serving performance and cannot be solved by only proactively dispatching.

Dynamic adapter-request co-migration. To address the load imbalance problem, we propose a adapter-request co-migration technique. The main idea is to migrate LoRA adapters and requests (with intermediate states) from overloaded replicas to others. Such migration reactively balances the load across replicas. As shown in Figure 8, replica 0 is overloaded with long-context requests while replica i is underloaded with short-context requests. The load imbalance can be mitigated by migrating some requests from replica 0 to replica i and loading corresponding adapters. However, it is not trivial to decide the optimal migration plan. Before introducing the algorithm in detail, we first model the reactive migration problem. The key notations are listed in Table 1.

Objective: overall running time. The goal of the migration algorithm is to minimize the overall running time among all replicas. The overall running time is determined by the placement of requests and LoRA adapters. We define two 0-1 matrices x and y to represent the placement of requests and LoRA adapters, respectively. Specifically, $x_{i,j}$ and $y_{k,j}$ are 0-1 variables that indicate whether request i or adapter k is in the replica j. Furthermore, we break x into two matrices x^G and x^H to represent the placement of requests in the GPU memory and host memory, respectively. The request with its

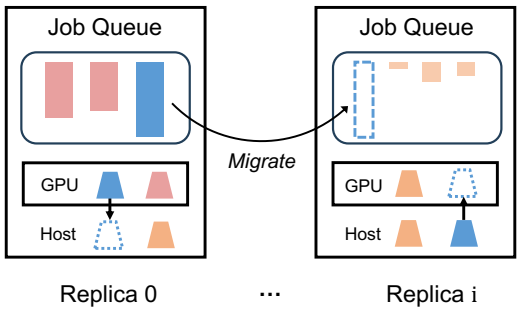


Figure 8: Dynamic migration.

Symbol	Description
$x^G_{i,j}$	Whether requests i is in the replica j's GPU memory.
$x^H_{i,j}$	Whether requests i is in the replica j's host memory.
$x_{i,j}$	Whether requests i is in the replica j.
$y_{k,j}$	Whether adapter k is in the replica j.
M^R_i	The memory consumption of request i.
M^A_k	The memory consumption of adapter k.
M^G_j	GPU memory capacity of replica j.
M^H_j	Host memory capacity of replica j.
L_i	Average exec time of requests with request i's adapter.
γ	Migration factor to control the request migration frequency.
η	Affinity factor to control the affinity between requests.

Table 1: Key notations in the migration problem.

intermediate results is either resident in the GPU memory or host memory (i.e., $x_{i,j} = x^G_{i,j} + x^H_{i,j} \leq 1$).

The overall running time of requests on each replica includes execution time and swapping overhead. On replica j, the execution time of request i is $L_i x_{i,j}$, where L_i is the average execution time of requests i. We scale the execution time by multiplying a migration factor γ to account for the migration time of request i. γ is calculated as the ratio of total executed time, including the migration time, to the total executed time without migration, inhibiting the frequent migration between replicas. As for swapping overhead, it is represented as $M^R_i x^H_{i,j}/B$, where M^R_i is the memory consumption and B is the PCIe bandwidth. As a result, the overall running time of replica j is estimated as $\sum_i \left(\gamma \times L_i x_{i,j} + M^R_i x^H_{i,j}/B\right)$. Because the dynamic batching algorithm prefers to batch requests with the same type, we also add a penalty term $\eta \sum_k y_{k,j}$ to represent the affinity among the same type of requests, where η is a hyperparameter. For all parallel replicas, the overall running time is the maximum running time of all replicas:

$$Min.\ \left(\max_j \sum_i (\gamma \times L_i x_{i,j} + M^R_i x^H_{i,j}/B) + \eta \sum_k y_{k,j}\right)$$

Adapter-request matching constraints. Different from classical load balancing problems [15–17], the adapter-request co-migration algorithm needs to consider both the migration of requests and the loading of LoRA adapters at the same time. One request can only be migrated to the replica only if there is a corresponding adapter available, which is formulated as following adapter-request matching constraints:

$$\forall i, j,\ y_{i,j} \geq x_{i,j}$$

Memory constraints. For each replica, the GPU memory consumption cannot exceed the GPU memory capacity. We define M^G_j as the GPU memory capacity and M^H_j as the host memory capacity of replica j. We get two constraints for GPU and Host memory, respectively:

$$\begin{cases} \forall j,\ \sum_i M^R_i \cdot x^G_{i,j} + \sum_k M^A_k \cdot y_{k,j} \leq M^G_j \\ \forall j,\ \sum_i M^R_i \cdot x^H_{i,j} \leq M^H_j \end{cases}$$

Existence constraints. Last but not least, the existence constraints ensure that each request is only placed in one replica, which is represented as:

$$\forall i,\ \sum_j x_{i,j} = 1$$

ILP formulation. Given the above objective and constraints, we formulate the co-migration problem as an ILP problem, where matrices x^G, x^H, and y are the variables. dLoRA solves the problem with the off-the-shelf ILP solver [28] and gets the optimal placement plan (i.e., x^G, x^H, and y). According to this plan, dLoRA migrates the intermediate states of requests and loads LoRA adapters simultaneously. dLoRA also reserves a chunk of memory in each replica, which is utilized to replicate additional LoRA adapters to maximize the burst tolerance as described in the proactive mechanism (§5.1). With this approach, dLoRA's co-migration algorithm effectively achieves load balancing across replicas with optimal performance.

Selective migration with constraints relaxation. The above ILP formulation offers an optimal migration plan, but its large decision space introduces computation overhead. As the GPU cluster and request number expand, the ILP's complexity increases exponentially. To address this, dLoRA employs two domain-specific heuristics to accelerate such solving process.

Our first insight is that migration is necessary primarily during extreme load imbalance. Therefore, dLoRA only triggers the migration algorithm when the available GPU memory of a replica is beyond a memory threshold or the queuing delay of requests in a replica is beyond a computation threshold. This approach effectively minimizes the migration frequency, thereby amortizing the migration overhead. Besides, similar to selective replication [15–17], dLoRA only considers migration between top K overloaded replicas and top K underloaded replicas. As a result, the complexity of the selective migration only depends on K rather than the cluster scale.

Our second insight is that the ILP problem can be simplified by relaxing the constraints. In practice, migration is implemented as token transfer and subsequent intermediate state reconstruction, which is much faster than direct intermediate state migration [12]. As a result, dLoRA does not differentiate

the requests on the GPU memory and host memory. Instead, dLoRA only considers the total memory consumption of requests and LoRA adapters. We assume that additional memory consumption is swapped out to the host memory which is always sufficient in practice. Therefore, memory constraints are negligible, with only the swapping overhead added to the total execution time.

Last, to reduce the complexity of the ILP problem, the variable matrix x is relaxed to the real number. Given the real number matrix x, the placement of each request is determined by the largest element in the corresponding row of x. For instance, request i is placed in replica j' if $x_{i,j'} = \max_j x_{i,j}$. With these techniques, dLoRA is able to solve the optimal adapter-request co-migration plan within milliseconds.

6 Implementation

We implement a prototype of dLoRA based on vLLM [12] with about 6.2K LOC. We use vLLM to build dLoRA as vLLM is the state-of-the-art serving system with advanced features such as PagedAttention and iteration-level scheduling. The prototype includes a FastAPI [29] frontend, a global scheduler, and GPU-based execution engines. The frontend of dLoRA extends the vLLM FastAPI [29] frontend, enables client-specific LoRA adapter selection per request. dLoRA's global load balancer is responsible for dispatching requests to LLM replicas and dynamically migrating requests. It uses Ray [30] actor to interact with execution engines in the cluster and utilizes Pulp [28] to solve the ILP problem defined in §5. The execution engine also uses Ray [30] actor for key-value cache management and LoRA adapter management. To support *unmerged inference* in §4, we transform the LoRA type of each request into a one-hot vector and generate a request-type mapping matrix of the current batch. We then utilize the `einsum` function provided by PyTorch to achieve parallel matrix multiplication and integrate it into the execution engine. We add LoRA adapters to the vLLM execution engine and swap adapters between GPU and host memory asynchronously to reduce overhead. As for model executor, we implement LoRA inference for popular LLMs, including OPT [31] and Llama-2 [2]. dLoRA also accomplishes Megatron-LM [32] style tensor parallelism on LoRA adapters to support distributed execution of large models which can not fit in a single GPU, e.g., Llama-2-70B. dLoRA can be integrated with other frameworks, such as Ray Serve [33]. On the server side, similar to HuggingFace PEFT [10], the cluster administrator needs to provide additional LoRA information, including LoRA adapter weights, LoRA adapter name and its dependency on the base LLM, before launching the service. dLoRA takes charge of other things. When sending a request to dLoRA, the client side specifies the LoRA adapter name in the request. dLoRA dispatches the request to the corresponding LoRA adapter and returns the result.

Model	Size	# of Layers	# of Heads	Hidden Size
Llama-2-7B	13GB	32	32	4096
Llama-2-13B	26GB	40	40	5120
Llama-2-70B	132GB	80	64	8192

Table 2: Model configurations.

7 Evaluation

In this section, we first demonstrate the end-to-end performance improvements of dLoRA over state-of-the-art LLM serving systems under diverse workloads and models. Then, we evaluate the design choices of dLoRA and show the effectiveness of each component.

7.1 Experiment Setup

Testbed. We evaluate dLoRA on a four-node GPU cluster, each with eight NVIDIA A800 80GB GPUs, i.e., 32 GPUs in total. Each node is equipped with 128 CPUs, 2048 GB of host memory, and a 200 Gbps InfiniBand NIC. We use PyTorch 12.1.0 and NVIDIA CUDA 12.2 for our evaluation.

Models. We choose the widely-used open-sourced LLMs, Llama-2 model series [2], as the pre-trained LLMs (i.e., base LLMs) for our evaluation. We use various Llama-2 models with different sizes, including Llama-2-7B, Llama-2-13B, and Llama-2-70B. The details of these models are shown in Table 2. The rank of LoRA adapters is set to 8, as used in the evaluation of prior work [8, 9].

Workloads. Similar to prior work [12], we generate workloads based on the ShareGPT dataset [26]. ShareGPT is a dataset collected from real-world conversations with OpenAI ChatGPT shared by users. We sample the arrival pattern of requests based on the production traces, Microsoft Azure function trace 2019 (MAF1) [34] and 2021 (MAF2) [25]. Although these traces are collected from Azure serverless functions, they are also widely used as the proxy of the LLM inference traces by prior work on model serving [13, 14]. Because the number of functions is larger than that of LLMs, we sort the functions based on the function invocation frequencies and map the functions to LoRA LLMs in a round-robin manner. To demonstrate different load patterns, we define the skewness as the number of each round-robin mapping. For example, if we map the first 10 functions to the first LoRA LLM, the second 10 functions to the second LoRA LLM, and so on, the skewness is 10. The larger the skewness, the more skewed the workload. To illustrate the impact of increasingly longer requests [35], we also scale the input and output length of requests by a scale ratio factor in §7.4.

Metrics. We use the average latency as the primary metric to evaluate the performance of dLoRA. Following prior work on LLM serving [11, 12], the average latency is calculated by dividing the sum of each request's end-to-end latency by the total number of output tokens. To compare different systems,

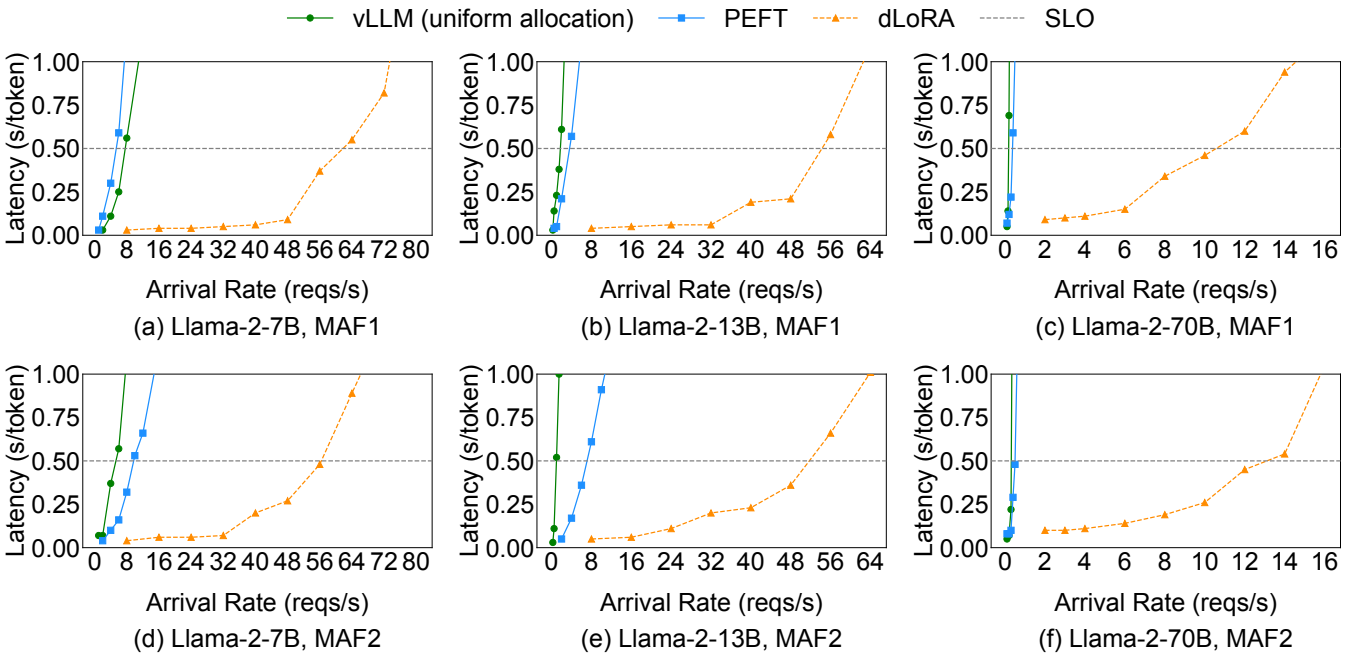


Figure 9: Average latency of different serving systems with Llama-2 models.

we set a latency service level objective (SLO) and compare the maximum throughput achieved by each system under the SLO. Similar to prior work, we set the latency SLO to 10× of the latency of a single iteration in the decoding phase. Specifically, SLO is 0.5 seconds based on our profiling.

Baselines. Because there is no existing system that specifically targets LoRA LLM serving. We compare dLoRA with two state-of-the-art LLM serving systems.

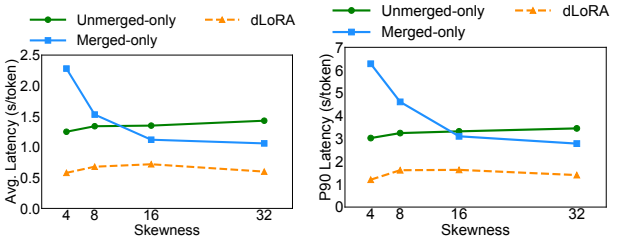
- **vLLM (uniform allocation)** [12]: vLLM is the state-of-the-art LLM serving system. It is a general-purpose LLM serving system, and ignores the multi-LLM serving scenario. To evaluate the performance of vLLM in the LoRA serving setting, we deploy multiple vLLM instances on the same cluster and uniformly allocate resources between them.
- **PEFT** [10]: PEFT is a HuggingFace library for parameter-efficient fine-tuning models. Although it can be used to serve LLMs, it does not support advanced features like selective batching [11] and PagedAttention [12]. To conduct a fair comparison, we implement these features for PEFT. PEFT batches requests based on their types and swap adapters between batches if necessary.

7.2 End-to-End Performance

We first compare end-to-end performance of dLoRA with vLLM (uniform allocation) and PEFT under MAF1 and MAF2 workload traces on Llama-2-7B, Llama-2-13B and Llama-2-70B models. The first column of Figure 9 shows the performance of these LLM serving systems when serving 128 Llama-2-7B models. Since Llama-7B is relatively small, the total GPU memory can accommodate full parameters of all models. However, vLLM (uniform allocation) and PEFT cannot dynamically batch different types of requests (i.e., unmerged inference) based on different workload patterns. In this case, dLoRA improves the throughput by up to 10.6× compared to vLLM (uniform allocation) and up to 11.5× compared to PEFT under the SLO requirement.

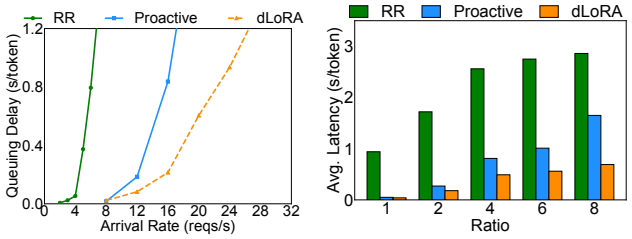
The second column of Figure 9 shows the performance of these LLM serving systems when serving 128 Llama-2-13B models. Since Llama-13B is larger than Llama-7B, the total GPU memory cannot accommodate full parameters of all models. As a result, vLLM (uniform allocation) which treats each LoRA LLM as an individual LLM has to swap the model parameters between host and GPU memory. Thus, dLoRA improves the throughput by up to 53.0× compared to vLLM (uniform allocation) under the SLO requirement. PEFT shares the base LLM between different LoRA LLMs to reduce memory footprint, but it does not have adapter-request migration. This significantly degrades the performance of PEFT. As a result, dLoRA improves the throughput by up to 15.0× compared to PEFT under the SLO requirement.

The third column of Figure 9 shows the performance improvement of dLoRA when serving 32 Llama-2-70B models in the cluster. Llama-2-70B is even larger than the GPU memory capacity of a single NVIDIA A800 80GB GPU. Therefore, the Llama-2-70B is partitioned across 4 GPUs with the tensor parallelism. As shown in Figure 9, the techniques employed by dLoRA integrate effectively with existing parallelism strategies and dLoRA improves the throughput by up



(a) Average Latency. (b) P90 Latency.

Figure 10: Effectiveness of the credit-based dynamic batching algorithm.



(a) Reduction in Queuing Delay. (b) Stability under Different Ratios.

Figure 11: Effectiveness of the adapter-request co-migration algorithm.

to 57.9× compared to vLLM (uniform allocation) and up to 26.0× compared to PEFT under the SLO requirement.

7.3 Effectiveness of Dynamic Batching

To show the effectiveness of dynamic batching with thresholds tuning, we compare the performance of dLoRA with strawman solutions described in §4. The experiments are conducted in a single NVIDIA A800 80GB GPU to serve 8 LoRA Llama-2-7B models, which avoids the effect of the adapter-request co-migration algorithm.

Figure 10(a) shows the average latency under diverse skewness. The arrival rate is set to make the average latency of dLoRA approximately equal to the SLO requirement. The first strawman solution, Merged-only, which always uses merged inference, performs poorly when the skewness is low, i.e., the type of requests is diverse, since it cannot serve different requests at the same time. Therefore, dLoRA improves the latency by up to 3.9×. The other strawman solution, Unmerged-only, which always uses unmerged inference, performs similarly no matter how the skewness changes. However, when the skewness is high, i.e., a few types of requests are dominant, Unmerged-only cannot take advantages of merged inference to avoid extra computation overhead caused by the adapter computation. Consequently, dLoRA improves the latency by up to 2.4× compared to Unmerged-only. In short, dLoRA dynamically switches between merged and unmerged inference based on the runtime workload, and always outperforms the two strawman solutions and achieves the optimal tradeoff.

In addition to the average latency, we also record the P90 latency of the three solutions with the same setting. As shown in Figure 10(b), dLoRA outperforms Merged-only and Unmerged-only by up to 5.2× and 2.5×, respectively. This consistent performance improvement indicates that although dLoRA may preempt some requests to serve other requests, the credit-based starvation prevention mechanism of dynamic batching improve performance while avoiding starvation.

7.4 Effectiveness of Dynamic Load Balancing

To show the effectiveness of the proactive and reactive dynamic load balancing, we compare the performance of dLoRA with two strawman solutions. We measure the average queuing delay of requests, an important indicator of load imbalance. Due to the limited budget, this experiment is conducted on a server with 8 NVIDIA A800 80GB GPUs. Figure 11(a) shows the experiment result. The first strawman solution is *RR* that directly dispatches requests to the corresponding LoRA LLMs preloaded by the workload-aware adapter placement in a round-robin manner without considering the bursty load imbalance (§5). Since RR does not consider the bursty load imbalance at all, dLoRA outperforms RR by 3.6× under the SLO requirement. The second strawman solution is *Proactive Dispatch* which only supports proactive mechanism in §5.1 without reactive migration. However, due to the variable and unpredictable length of requests, Proactive Dispatch cannot handle this unpredictable run-time load imbalance. In contrast, dLoRA dynamically migrates requests between replicas to balance the load and outperforms Proactive Dispatch by 1.4× under the SLO requirement.

We also evaluate the stability of the dynamic load balancing algorithm under different scale ratio factors of input and output length of requests. Figure 11(b) shows the average latency of requests under different ratios. Because RR cannot dynamically change LoRA adapter distribution across replicas as the ratio increases, the average latency of requests increases rapidly, and thus the average latency of RR is up to 23.5× higher than dLoRA. Although Proactive Dispatch Only is able to dynamically load LoRA adapters and dispatch the requests to mitigate load imbalance (i.e., it outperforms RR by up to 6.5×), it is not able to migrate requests between replicas to handle increasingly variable requests, leading to unpredictable load imbalance. As a result, the average latency of Proactive Dispatch is up to 2.39× higher than dLoRA.

7.5 Scalability and Overhead

For a cluster-scale serving system, we conduct evaluations to analyze the scalability of dLoRA. Figure 12 shows the throughput of dLoRA under different numbers of adapters. We increase the number of adapters increases while keeping the arrival rate constant. As the number of adapters increase, dLoRA achieves stable throughput. However, vLLM (uniform allocation) faces scalability challenges because it has to main-

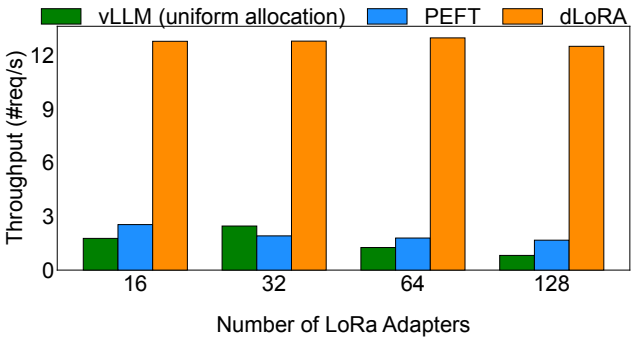


Figure 12: Scalability of dLoRA.

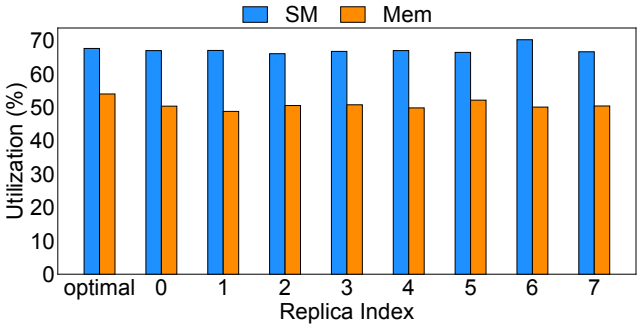


Figure 13: GPU utilization of dLoRA.

tain separate instances of full LLM weights for each LoRA LLM whcih incurs high memory footprint. PEFT struggles to scale efficiently as well because it cannot dynamically batch different types of requests. Moreover, the two baselines cannot manage the load imbalance between replicas. As a result, the throughput of vLLM (uniform allocation) and PEFT decreases by up to 3.0× and 1.5× respectively, as the number of adapters increases to 128.

We also evaluate the runtime overhead of dLoRA. Figure 15 breaks down the total latency of dLoRA's inference into three parts: the solving time of the ILP solver for the dynamic co-migration algorithm, the switching overhead of the dynamic batching algorithm, and the actual inference time. As shown in this figure, the overhead introcduced by dLoRA, i.e., ILP solving time and the switching overhead, are negligible compared to the actual inference time. The actual inference time consistently accounts for over 96.7% of the total latency, regardless of the arrival rate. The negligible runtime overhead of dLoRA mainly comes from the fact that dLoRA migrates requests or switches inference mode occasionally and the overhead is amortized by a number of iterations.

7.6 GPU Utilization

To show the GPU utilization, we measure the streaming multiprocessor (SM) utilization and GPU memory utilization of each replica in the cluster.The setup is the same as §7.4. dLoRA serves the Llama-2-13B model. The request rate is

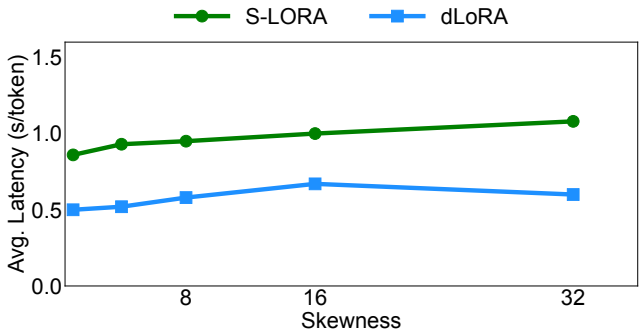


Figure 14: Comparison with SLoRA.

set to make the average latency of dLoRA approximately equal to the SLO requirement. We also measure the optimal SM utilization and GPU memory utilization when all requests belong to the same type, and vLLM always uses merged inference. As shown in Figure 13, although requests belong to different LoRA types, dLoRA achieves nearly the same SM utilization and GPU memory utilization compared to the optimal utilization, which indicates that dLoRA effectively utilizes the GPU resources and avoids resource waste as shown in Figure 2(a). Besides, the SM utilization and GPU memory utilization among replicas are balanced, which shows the effectiveness of the dynamic load balancing algorithm.

7.7 Comparison with Concurrent Work

S-LoRA [18] is a concurrent work for serving multiple LoRA LLMs. S-LORA also tries to batch requests destined for different LoRA adapters. However, S-LoRA does not consider the load imbalance between replicas and is not able to dynamically switch between merged and unmerged inference. The parallelism strategy proposed by S-LoRA is also orthogonal to dLoRA. Figure 14 compares dLoRA with S-LoRA using the same setting as §7.3. As shown in the figure, no matter how the skewness changes, dLoRA consistently outperforms S-LoRA by up to 1.8× in terms of average latency. The reason is that S-LoRA statically serves requests by unmerged-only inference, which ignores the opportunity of using merged inference to reduce the computation overhead. In contrast, dLoRA exploits this opportunity and is more effective in handling diverse requests with different LoRA types.

8 Related Work

LLM serving systems. Recently, many works have been proposed for LLM serving. Orca [11] proposes iteration-level scheduling to continuously batch requests with different lengths at the iteration level. vLLM [12] proposes a PagedAttention operator and a block-based KV cache management mechanism to reduce GPU memory fragmentation. FastServe [19], DeepSpeed-FastGen [36], and SARATHI [37] leverage the characteristics of LLM serving to schedule requests with different lengths. Nvidia FasterTransformer [38],

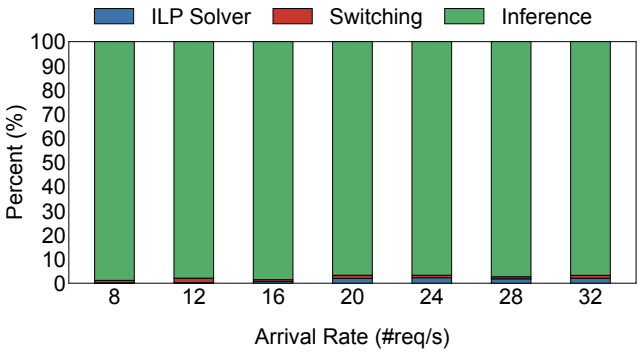


Figure 15: Overhead of dLoRA.

DeepSpeed Inference [39], and the Google serving system for PaLM [40] optimize the GPU/TPU implementation and parallelism specifically for LLM inference. FlexGen [41] uses offloading to improve the throughput of LLM serving. SpecInfer [42] uses speculative decoding to reduce the latency of LLM serving systems. SpotServe [43] tries to leverage spot instances to reduce the cost of LLM serving. These works focus on the optimization of a single LLM. In contrast, dLoRA is a serving system for multiple LoRA LLMs in a GPU cluster.

S-LoRA [18] and Punica [44] are concurrent works that also propose to serve multiple LoRA LLMs by batching requests destined for different adapters. S-LoRA proposes a new parallelism strategy, which is orthogonal to dLoRA. Punica uses an ad-hoc migration strategy to reduce the number of used GPUs, which is also different from dLoRA. dLoRA proposes dynamic load balancing algorithms to balance the load among GPUs, which is not considered in Punica and S-LoRA. They also overlook the opportunity of using merged inference to further reduce the latency of LLM serving by only adopting a merged-only inference strategy.

Traditional DNN serving systems. Many production-ready DNN serving systems have been developed, such as Tensorflow Serving [45] and Triton Inference Server [46], but they do not have LLM-specific optimizations. Some recent DNN serving systems, such as Clipper [47], ClockWork [48], SHEPHERD [13], Tabi [49], and Paella [50] serves multiple DNN models in cluster wide, but they mainly focus on small DNN models, such as ResNet and BERT. AlpaServe [14] leverages diverse parallelism strategies to accelerate serving multiple large DNN models in a GPU cluster, but it does not target autoregressive LLM serving and LoRA models. PetS [24] considers the scenario of serving multiple parameter-efficient DNN models in a GPU server, but it does not consider serving autoregressive LLMs in a GPU cluster and the unique system characteristics of LoRA adapters. DVABatch [51] uses multi-entry multi-exit batching to serve diverse models simultaneously, but it does not target either LLMs or LoRA.

Load balancing. Load balancing has been studied in many research fields, such as networking and cloud computing. Many load balancers, such as Ananta [52], Beamer [53] and Maglev [54], try to improve the performance of dispatching packets, but as discussed above, it is not sufficient to solve load imbalance in our scenario. Other kinds of load balancers, such as Pegasus [15], Scarlett [16] and E-Store [55], adopt selective replication or migration to balance the load. However, they are not suitable for our scenario, and the dependency between requests and adapters makes the load imbalance in our scenario even more complicated.

9 Conclusion

We present dLoRA, an inference serving system for LoRA models. dLoRA dynamically orchestrates requests and adapters for efficient LLM serving in each worker replica and across worker replicas. For each replica, dLoRA employs a dynamic batching technique to leverage both merged and unmerged inference to improve the efficiency of LLM serving. For the cluster, dLoRA employs a dynamic load balancing technique to migrate both requests and adapters to balance the load among worker replicas. Based on these techniques, we build a system prototype of dLoRA. Evaluation on production traces shows that dLoRA achieves up to 57.9× and 26.0× higher throughput than vLLM and HugginFace PEFT. dLoRA also achieves up to 1.8× lower average latency than the concurrent work S-LoRA.

Acknowledgments. We sincerely thank our shepherd and the anonymous reviewers for their valuable feedback. This work was supported by the National Natural Science Foundation of China under the grant number 62325201, 62172008, and the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Xuanzhe Liu is the corresponding author. Bingyang Wu, Ruidong Zhu, Zili Zhang, Xuanzhe Liu and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] OpenAI, "GPT-4 technical report," 2023.

[2] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv*, 2023.

[3] B. Roziére, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code Llama: Open Foundation Models for Code," *arXiv*, 2023.

[4] "OpenAI Documentation: Fine-tuning." https://platform.openai.com/docs/guides/fine-tuning, 2023.

[5] "Announcing Anyscale Private Endpoints and Anyscale Endpoints Fine-tuning." https://www.anyscale.com/blog/announcing-anyscale-private-endpoints-and-anyscale-endpoints-fine-tuning, 2023.

[6] "Customize a model with Azure OpenAI Service." https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/fine-tuning?tabs=turbo%2Cpython&pivots=programming-language-studio, 2023.

[7] "Fine-Tuning Llama 2 with Together.ai: A Step-by-Step Guide." https://blog.gopenai.com/fine-tuning-llama-2-with-together-ai-a-step-by-step-guide-cf2f3cce659d, 2023.

[8] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-Rank Adaptation of Large Language Models," in *ICLR*, 2022.

[9] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient Finetuning of Quantized LLMs," *arXiv*, 2023.

[10] S. Mangrulkar, S. Gugger, L. Debut, Y. Belkada, S. Paul, and B. Bossan, "PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods." https://github.com/huggingface/peft, 2022.

[11] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A Distributed Serving System for Transformer-Based Generative Models," in *USENIX OSDI*, 2022.

[12] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *ACM SOSP*, 2023.

[13] H. Zhang, Y. Tang, A. Khandelwal, and I. Stoica, "SHEPHERD: Serving DNNs in the Wild," in *USENIX NSDI*, 2023.

[14] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, *et al.*, "AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving," in *USENIX OSDI*, 2023.

[15] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports, "Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories," in *USENIX OSDI*, 2020.

[16] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters," in *EuroSys*, 2011.

[17] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *USENIX NSDI*, 2013.

[18] S. Ying, C. Shiyi, L. Dacheng, H. Coleman, L. Nicholas, Y. Shuo, C. Christopher, Z. Banghua, Z. Lianmin, K. Kurt, *et al.*, "S-LoRA: Serving thousands of concurrent lora adapters," *Conference on Machine Learning and Systems*, 2023.

[19] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, "Fast Distributed Inference Serving for Large Language Models," in *arXiv*, 2023.

[20] X. Wang, Y. Xiong, Y. Wei, M. Wang, and L. Li, "LightSeq: A High Performance Inference Library for Transformers," in *NAACL*, 2021.

[21] A. Chavan, Z. Liu, D. Gupta, E. Xing, and Z. Shen, "One-for-All: Generalized LoRA for Parameter-Efficient Fine-tuning," *arXiv*, 2023.

[22] Y. Chen, S. Qian, H. Tang, X. Lai, Z. Liu, S. Han, and J. Jia, "LongLoRA: Efficient fine-tuning of long-context large language models," *arXiv*, 2023.

[23] T. Gong, C. Lyu, S. Zhang, Y. Wang, M. Zheng, Q. Zhao, K. Liu, W. Zhang, P. Luo, and K. Chen, "Multimodal-GPT: A vision and language model for dialogue with humans," *arXiv*, 2023.

[24] Z. Zhou, X. Wei, J. Zhang, and G. Sun, "PetS: A unified framework for Parameter-Efficient transformers serving," in *USENIX ATC*, 2022.

[25] Y. Zhang, I. n. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *ACM SOSP*, 2021.

[26] "ShareGPT Teams." https://sharegpt.com/, 2023.

[27] "Introducing ChatGPT." https://openai.com/blog/chatgpt, 2022.

[28] "PuLP: A Python Linear Programming API.." https://github.com/coin-or/pulp, 2009.

[29] "FastAPI Documentation.." https://fastapi.tiangolo.com/, 2018.

[30] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *USENIX OSDI*, 2018.

[31] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "OPT: Open Pre-trained Transformer Language Models," *arXiv*, 2022.

[32] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *arXiv*, 2020.

[33] "Ray serve: Scalable and programmable serving." https://docs.ray.io/en/latest/serve/index.html, 2023.

[34] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *USENIX ATC*, 2020.

[35] "Introducing Claude 2.1." https://www.anthropic.com/index/claude-2-1, 2023.

[36] "DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference." https://github.com/microsoft/DeepSpeed/tree/master/blogs/deepspeed-fastgen, 2023.

[37] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills," 2023.

[38] N. Corporation, "FasterTransformer." https://github.com/NVIDIA/FasterTransformer, 2019.

[39] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, and Y. He, "DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale," in *SC*, 2022.

[40] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Conference on Machine Learning and Systems*, 2023.

[41] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Re, I. Stoica, and C. Zhang, "FlexGen: High-throughput Generative Inference of Large Language Models with a Single GPU," *International Conference on Machine Learning (ICML)*, 2023.

[42] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, Z. Wang, R. Y. Y. Wong, A. Zhu, L. Yang, X. Shi, C. Shi, Z. Chen, D. Arfeen, R. Abhyankar, and Z. Jia, "SpecInfer: Accelerating Generative Large Language Model Serving with Speculative Inference and Token Tree Verification," 2023.

[43] P. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, and Z. Jia, "SpotServe: Serving Generative Large Language Models on Preemptible Instances," *ACM ASPLOS*, 2024.

[44] L. Chen, Z. Ye, Y. Wu, D. Zhuo, L. Ceze, and A. Krishnamurthy, "Punica: Multi-Tenant LoRA Serving," *Conference on Machine Learning and Systems*, 2023.

[45] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "Tensorflow-serving: Flexible, high-performance ml serving," *arXiv*, 2017.

[46] N. Corporation, "Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution.." https://github.com/triton-inference-server/server, 2019.

[47] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency Online Prediction Serving System.," in *USENIX NSDI*, 2017.

[48] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up," in *USENIX OSDI*, 2020.

[49] Y. Wang, K. Chen, H. Tan, and K. Guo, "Tabi: An Efficient Multi-Level Inference System for Large Language Models," in *EuroSys*, 2023.

[50] K. K. W. Ng, H. M. Demoulin, and V. Liu, "Paella: Low-Latency Model Serving with Software-Defined GPU Scheduling," in *ACM SOSP*, 2023.

[51] W. Cui, H. Zhao, Q. Chen, H. Wei, Z. Li, D. Zeng, C. Li, and M. Guo, "DVABatch: Diversity-aware Multi-Entry Multi-Exit batching for efficient processing of DNN services on GPUs," in *USENIX ATC*, 2022.

[52] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri, "Ananta: Cloud Scale Load Balancing," in *ACM SIGCOMM*, 2013.

[53] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless Datacenter Load-balancing with Beamer," in *USENIX NSDI*, 2018.

[54] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu,

B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A Fast and Reliable Software Network Load Balancer," in *USENIX NSDI*, 2016.

[55] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, "E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems," *Proceedings of the VLDB Endowment*, 2014.

A Appendix

A.1 Starvation Prevention of Algorithm 1

Algorithm 3 shows the pseudo-code of the credit-based dynamic batching algorithm to prevent starvation. Initially, we assign the same credit to each LoRA adapter, which guarantees the fairness in the beginning. Whenever a request is not served in the FCFS order, the credit of the corresponding LoRA adapter needs to be transferred to the requests that orignially should be served in the FCFS order (line 14 and line 21). In this, way, if a LoRA adapter is not served for a long time, it will accumulate enough credit and our algorithm always serves them first (lines 11–15). We also prevent the LoRA adapter to merge into the base LLM weights when it does not have sufficient credit (line 18). To mitigate the oscillation of starvation prevention between different LoRA adapters, we set a threshold T_{starve} to decide the starvation (lines 6–10). This parameter can be tuned to make a tradeoff between performance and fairness.

Algorithm 3 Credit-based Dynamic Batching

1: **function** CREDITBATCHING(B_{fcfs}, R, S, L)
2: **Input:** FCFS requests B_{fcfs}, Request $R = \{r_1, r_2, ..., r_n\}$
3: Replica state S, LoRA adapters $L = \{l_1, l_2, ..., l_m\}$
4: **Output:** The batch of requests to be executed B_{next}
5: // Stavation Prevention
6: **for** $l \in L$ **do**
7: **if** $l_i.state == S_{starve} \wedge l_i.credit < T_{normal}$ **then**
8: $l_i.state = S_{normal}$
9: **else if** $l_i.credit > T_{starve}$ **then**
10: $l_i.state = S_{starve}$
11: $L_{starve} = \{l_i \in L \mid l_i.state == S_{starve}\}$
12: $B_{starve} = \{r_i \in R \mid r_i.type \in L_{starve}\}[: max\_bs]$
13: **if** $|B_{starve}| > 0$ **then**
14: $transfer\_credit(B_{starve}, B_{fcfs})$
15: **return** $B_{next} = B_{starve}$
16:
17: // Adaptive switching between different modes
18: $L_{eligible} = \{l_i \mid l_i.creidt \geq credit(\{r_i \mid r_i.type == l_i\}, B_{fcfs})\}$
19: RET =DYNAMICBATCHING($B_{fcfs}, R, S, L_{eligible}$)
20: **if** $RET\ != B_{fcfs}$ **then**
21: $transfer\_credit(RET, B_{fcfs})$
22: **return** $B_{next} = RET$