EcoServe: Maximizing Multi-Resource Utilization with SLO Guarantees in LLM Serving

Haiying Shen and Tanmoy Sen
Department of Computer Science, University of Virginia
Email:hs6ms,ts5xm @virginia.edu

Abstract

Large Language Model (LLM) inference confronts an inherent problem of GPU memory bottleneck, resulting in low GPU utilization and throughput. Previously proposed LLM job schedulers fail to fully utilize GPU and memory at each iteration or ensure Key-Value cache (KVC) allocations upon request. In this work, we first adopted a job scheduler for the cloud and found it can mitigate instead of completely address the problems but greatly increase the scheduling time. To address these problems, we perform a trace-based experimental analysis, leading to the proposal of a time-Synced Batching system with Decoupled request processing and KVC pipelining (EcoServe). To enable adding prompts to a batch to maximize GPU utilization in each iteration, EcoServe maintains separate waiting queues for prompt and generation tasks (GT). It batches GTs with the same predicted response lengths (RL) to save scheduling time and allocates KVC space for the predicted RL to avoid KVC allocation failures. It prioritizes queued requests that occupy more KVC to release KVC earlier and satisfy request service-level-objective (SLO). EcoServe further has a novel KVC pipelining method, allowing the sharing of allocated but unused KVC space to enhance KVC utilization. Experimental results demonstrate that EcoServe increases throughput by up to 4× with the same level of latency, generates up to 91% lower job completion time and up to 91% higher SLO satisfaction ratio compared to the vLLM state-of-the-art scheduler. EcoServe'code has been open-sourced.

1 Introduction

Large Language Models (LLMs) have emerged as transformative tools widely used in various applications such as natural language understanding, machine translation, text summarization, question answering, and sentiment analysis. In LLM inference, a user's request undergoes two primary stages: computation-intensive prompt processing task (PT) and memory-intensive token generation task (GT). The first token is generated by the PT in the first iteration, and then GT utilizes the newly generated token as input to produce the next token in the response sequence in each iteration. During these iterations, each token's key-value (KV) pairs are stored in the KV cache (KVC) in GPU memory and are accessed during the calculation of subsequent tokens. This iterative process continues until the request is completed, and then its occupied KVC is released.

Table 1. Comparison of EcoServe and current methods.

Method	Avoid KVC overflow	Increase GPU uti. when KCV allows	High GPU uti. in each iteration	High KVC uti. in each iteration	Low scheduling time
Orca [11]	✓	×	×	×	✓
FastServe [12]	✓	×	×	×	×
vLLM [13]	×	×	×	✓	✓
FastGen [14]	×	✓	×	×	✓
Sarathi-Serve [15]	×	✓	×	×	✓
EcoServe	✓	✓	✓	✓	✓

To meet the capacity requirements of different applications, LLMs have experienced significant growth in size. Early models like BERT [1] had around 340 million parameters, while GPT-3 and GPT-4 [2, 3] reached 175 billion. GPT-4o [4] exceeds 200 billion parameters and LLama 3.1 [5, 6] reaches 405 billion parameters. The increasing size and widespread use of machine learning (ML) inference applications have led to the reliance on expensive, power-hungry GPUs, making model inference a significant operational cost for production clusters [7]. For instance, Facebook processes over 2 quadrillion inference requests daily [7], inference accounts for more than 90% of ML production costs on AWS[8], and operating ChatGPT incurs approximately $700,000 per day in compute hardware costs for 28,936 GPUs [9]. Projections suggest that LLM-related costs for data centers could exceed $76 billion by 2028, driven by the rapid expansion of GPU deployments [10].

To reduce the cost of LLM serving systems, in this paper, our goal is to design a scheduler that *fully utilizes both GPU compute and memory resources (dual-resources) in each iteration while meeting service-level objectives (SLOs), minimizing scheduling time (i.e., batch formation time) and preventing KVC overflow.* To the best of our knowledge, no existing schedulers have achieved this goal (see Table 1).

Different schedulers have been proposed and many schedulers use first-come-first-serve (FCFS). To improve request-level scheduling [16, 17], Orca [11] employs iteration-level scheduling that returns completed requests after each iteration and selects waiting requests to form a new batch. It allocates a request with KVC space for the maximum total sequence length, i.e., the sum of the prompt length and response length (RL), to prevent cache overflow [18]. However, this *max-allocation* approach leads to low KVC utilization due to unutilized allocated KVC and limits batch size, resulting in low GPU utilization (potentially as low as 0.4%

[19]) and throughput. FastServe [12] employs the traditional Multi-Level Feedback Queue (MLFQ) [20] to address issues of head-of-line blocking and prolonged JCT, but also uses max-allocation. Subsequent work has focused on improving either KVC utilization or GPU utilization, as outlined below.

Improve KVC utilization. To mitigate this KVC bottleneck, vLLM [13] allocates fixed-size KVC blocks (e.g., 32 tokens) to a request that starts execution or uses up its previously allocated block (we call it *block-allocation*), and preempts requests upon KVC overflow in execution by swapping their KV values to CPU memory or conducting recomputation. However, it cannot guarantee KVC allocations upon request in execution, generating preemptions and delay.

Improve GPU utilization. FastGen [14] and Sarathi-Serve [15] chunk long prompts and batch PTs with GTs to reach the target forward size (TFS) that maximizes GPU utilization. Forward size is the number of tokens in a batch [21, 22]. However, by employing block-allocation, they inherit its shortcomings mentioned above.

To avoid KVC overflows, we could adopt the approach in [23] that allocates the KVC equal to the estimated RL, referred to as *exact-allocation*. With exact-allocation, each waiting prompt's memory-demand can be estimated, and we can choose prompts to fully allocate the dual-resources. Note that the allocated compute resource will be fully utilized while the allocated KVC won't be fully utilized. A batch is formed by selecting requests until the KVC is fully allocated. After each iteration, the allocated KVC for a prompt with length L_p is retained until completion, while the compute resources for $(L_p - 1)$ tokens are released. However, no new requests can be added to the batch to utilize the released compute resources, leading to low GPU utilization. We call such an issue *GT domination issue*. We verified this issue through trace-based experimental analysis, which also revealed that:

1) Varying dual-resource demands of prompts makes it difficult to find prompts that fully allocate dual-resources concurrently.
2) Identifying prompts in the waiting queue to fully allocate the dual-resources incurs a long scheduing time.
3) Exact-allocation still leads to underutilizations of both GPU and KVC (dual-resources) due to unutilized allocated KVC and limited batch size.
4) To address the KVC under-provisioning using LLM-based RL prediction [23], there exists a sweetspot padding ratio to the predicted value that balances execution time and waiting time. In addition, upon a KVC allocation failure, using the reserved KVC and preemption without KVC offloading to CPU memory (offload-free preemption) may be more efficient than vLLM's offload-based preemption

many modern GPU clusters for LLMsare equipped with InfiniBand??

To address the issues and leverage the observations, we propose EcoServe system, which employes exact-allocation, to maximize multi-resource Utilization with SLO guarantees. EcoServe consists of the following components:

- **KVC Pipelining Utilization.** Each GT lends its unused allocated KVC to another GT, carefully selecting a recipient to ensure that by the time the original GT requires the lent KVC, the recipient has completed and released it. The recipient GT follows the same process and so on. This iteration continues until the original GT's allocated KVC is fully utilized.
- **Time-Synced Batching with Decoupled Dual-Resource Allocation (*SyncDecoupled*)**. It does not account for the KVC demand of a prompt's output, leaving PTs and GTs responsible for fully utilizing the GPU and KVC for each iteration, respectively. It tries to synchronize the completion time of requests in a batch by batching the requests with the same predicted RL to avoid iteration-level scheduling. To account for RL under-prediction, it uses padding on predicted values, reserving KVC, and offload-free preemption.
- **Prompt and Generation Task Ordering.** It orders the waiting PTs and waiting GTs. It first tries to ensure compliance with JCT service-level-objective (SLO) requirements, then prioritizes the tasks that occupy a larger KVC space so they can release KVC earlier, and finally prioritizes the tasks with longer prompt lengths or predicted RLs to quickly find tasks to fully utilize the resources.

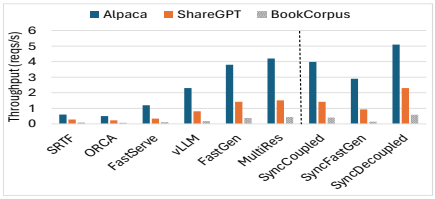
Unlike methods that disaggregate GTs and PTs of a request across different GPUs [24–26], which are suitable for high node-affinity clusters with Infiniband (for KV value transfer) and abundant GPU resources (for hosting multiple model copies), EcoServe operates without such constraints. It focuses on fully utilizing GPU dual-resources to achieve cost-efficiency – critical given the limited availability of GPUs in institutions today. Furthermore, many smaller institutions and company, including the authors' department, have clusters that utilize only Ethernet connections instead of Infiniband. In contrast to [23], which batches requests with similar output lengths to prevent delaying faster requests in the request-level scheduling, we examine how this method reduces scheduling time at the iteration-level scheduling for more complex schedulers that fully utilize dual-resources, rather than relying on FCFS-based schedulers.

The contribution of this work includes:

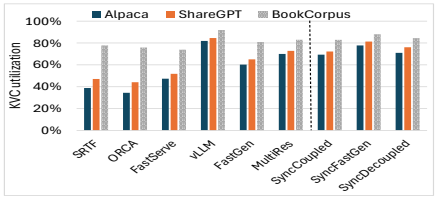
- An in-depth trace-based experimental analysis that lays the foundation of the system design;
- A novel EcoServe system that achieves the aforementioned goal, which fail to achieve by previous methods as shown in Table 1;
- Real implementation of EcoServe and a comprehensive trace-driven performance evaluation.

Our experiments show that EcoServe increases throughput by up to 4× with the same level of latency, generates up to 91% lower job completion time (JCT) and up to 91% higher SLO satisfaction ratio compared to the vLLM state-of-the-art scheduler.

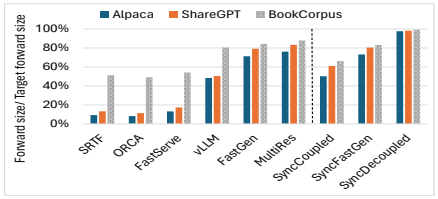
??



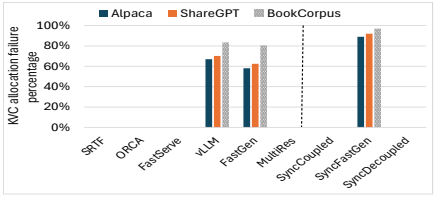
(a) Throughput.



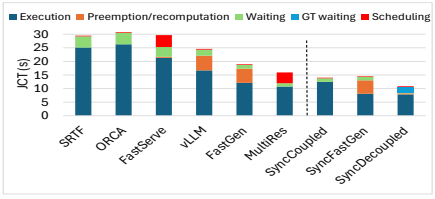
(b) KVC utilization.



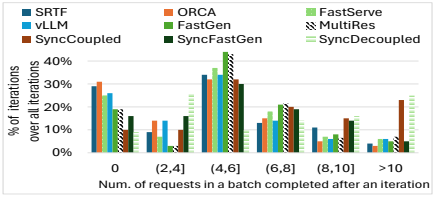
(c) Forward size.



(d) KVC allocation failure percentage.



(e) JCT for Alpaca.



(f) Distribution of iterations with a certain num. of requests completed after an iteration for Alpaca.

Figure 1. Comparison of different schedulers.

2 Experimental Analysis

2.1 Experiment Settings

Machine settings. We ran our experiments on an AWS p4d.24xlarge instance, equipped with 8 NVIDIA A100 GPUs and 1152GB of total CPU and GPU memory. Each GPU is equipped with 80GB of memory. The GPUs are connected with a 600 GB/s NVSwitch. We ran the OPT model with 13B parameters [27] on one GPU. The memory for KVC is 12GB.
Request settings. We used the Alpaca [28], ShareGPT [29] and BookCorpus [30] traces. Table 2 shows the properties of the traces and our settings. For each trace, we used 10K requests for fine-tuning the RL prediction model, and the rest of the requests for experiments. The batch size of Orca and FastServe was set to 8 [11]. We set TFS by finding the forward size that saturates GPU utilization as in [14]. As in [13], the block size was set to 32 and the request arrival rate followed a Poisson distribution. We set the arrival rates for each dataset as indicated in Table 2 to create our scenario that some requests are queued when a batch is processing. We divided the prompts in BookCorpus [30] into 2048-token chunks to meet the requirement of our used LLM model. The KVC utilization (measured using the gpustat library [31]) was collected at a 1s time interval.

Table 2. Trace properties and experiment settings.

Trace	Input length			Output length			Req	Arrival rate
	avg	min	max	avg	min	max		
Alpaca	19.31	9	2.47K	58.41	13	292	52K	36 reqs/s
ShareGPT	161.31	16	3.2K	337.99	19	991	90K	28 reqs/s
BookCorpus	1952.11	18	461K	681.2	32	1041	11K	1.2 reqs/s

Schedulers. We assumed that the RL of each request is pre-known in our first measurement, and evaluated the following schedulers: 1) Orca; 2) The shortest-remaining-time-first in iteration-level scheduling using max-allocation (SRTF); 3) FastServe [12] using 5-level queues; 4) vLLM [13] with the KVC swapping strategy; and 5) FastGen [14] with the recomputation strategy.

To improve dual-resource utilization, we naturally adopt a multi-resource scheduler [32] for the cloud that identifies queued requests which will fully utilize the dual-resources concurrently at each iteration (called *MultiRes*). It uses exact-allocation and calculates the Euclidean distance between each request's demands on GPU and KVC and the available resources and chooses the request with the minimum Euclidean distance, and repeats this process after each request selection until no requests can be added to the batch. Its time complexity for scheduling is $O(n^2)$, where n is the number of requests in the queue.

We implemented Orca and FastServe ourselves since their source codes are not publicly available. We used the source code of FastGen [14], and implemented the other schedulers based on vLLM's source code [13]. The error bars in our reported results are the 5th percentile and the 95th percentile. For figures with three datasets, unless specified otherwise, we discuss the average value of the three datasets.

2.2 Motivation and Exploration for a New Scheduler

A request's *waiting time* is the period its prompt waits in the queue before it starts to execute, its *preemption time* is the period it pauses running, and its *execution time* is the period from when it is dispatched to the execution engine to when it is completed, excluding the scheduling time and preemption time, and GT queuing time in EcoServe. Figure 1 shows the performance in different metrics of different schedulers (we will introduce the right three schedulers later). The JCT

is decomposed into different stages. Due to space limits, we only present the last two figures for Alpaca. Results for the other two datasets show similar patterns and will be presented in the appendix.

SRTF, Orca, and FastServe underperform vLLM in terms of throughput, KVC utilization, forward size, and JCT. Their worse performance stems from the max-allocation, which results in KVC bottleneck, and limits the batch size and GPU utilization. However, as shown in Figure 1d, using block-allocation, vLLM and FastGen generate 74% and 67% KVC allocation failure percentages. The resultant preemptions or recomputations cause additional delays, constituting 20% and 16.5% of the JCT in vLLM and FastGen, respectively, as shown in Figure 13a.

In batching, vLLM fully allocates KVC while FastGen aims to fully utilize GPU by reaching TFS, so vLLM generates 20% higher KVC utilization, 41% lower throughput, 27% lower forward size, and finally 29% higher JCT than FastGen. We see that even though FastGen aims to reach TFS, it cannot achieve it in each iteration due to the limit of KVC.

To address the problems, *MultiRes* aims to fully utilize both GPU and KVC and also uses exact-allocation. The exact-allocation helps *MultiRes* eliminate the KVC overvelows as shown in Figure 1d, but since the allocated KVC is not always used (i.e., reserved waste [13]), it generates similar KVC utilization as FastGen. *MultiRes* improves the throughput of FastGen by 10% and its forward size by 5%. Consequently, the JCT of *MultiRes* is 42.63% lower than vLLM and 35% lower than FastGen. Simple FCFS generates negligible scheduling time, while FastServe and *MultiResc* generate scheduling time occupying 17% and 34% of their JCT, respectively. Comparing the results of the three datasets, we found BookCorpus generates lower throughput, higher KVC utilization, more KVC allocation failures due to its longer prompts.

Surprisingly, *MultiRes* does not improve the throughput and JCT greatly and fails to maximize GPU utilization in each iteration. The reasons are explained in §1. To verify the inherent GT domination issue, we measured the distribution of iterations that have a certain number of requests completed, as shown in Figure 1f. We see that not every iteration will complete some requests and the number of completed requests in an iteration is limited. SRTF, Orca, FastServe, vLLM, FastGen and *MultiRes* have 27%-35% iterations with 0 completed requests for the three datasets, and have 4%-14% iterations with (2,4] completed requests. Then, zero or few PTs can be added to the batch, making it difficult to achieve TFS.

Observation 1. *MultiRes* still cannot fully utilize GPU or KVC due to the inherent GT domination issue, its coupled scheduling nature that considers each request for fully utilizing the dual-resources and the reserved waste. In addition, it incurs high scheduling time (e.g., 34% of JCT).

To reduce scheduling time, we aim to avoid heavyweight iteration-level scheduling. Hence, we group requests with synchronized completion times, i.e., same-RL requests, and allocate groups to the batch until the KVC is fully utilized. We call this method *SyncCoupled*. *MultiRes* can be considered as *UnsyncCoupled*. The feasibility of *SyncCoupled* depends on whether there are some same-RL queued requests that can be grouped. Figure 2 shows the CDF of groups versus the number of requests in one GT group in a batch for all three traces in *SyncCoupled*. For Alpaca and ShareGPT, 20% and 23% of the groups have at least 12 requests and 61% and 59% of the requests have at least 4 requests. For BookCorpus, 22% of the groups have at least 2 requests in a batch.

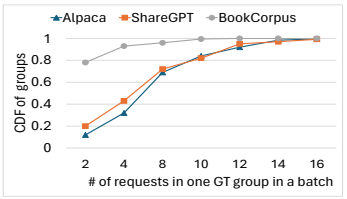


Figure 2. Num. of requests in a same-RL group in experiment.

Observation 2. Grouping same-RL requests is feasible in request processing.

The performance of *SyncCoupled* is also included in Figure 1. Compared to *MultiRes*, *SyncCoupled* generates scheduling time constituting only 2.14% of JCT, but reduces throughput by 7.2%. However, due to the inherent GT domination issue, KVC is unlikely to have the space to enable adding prompts to increase GPU utilization. Even if we can add prompts, they are unlikely to complete at the same time as the current requests in the batch, which conflicts with the time-sync principle of *SyncCoupled*. To mitigate the KVC limitation, we could use the block-allocation as in FastGen [14] (we call this method *SyncFastGen*). The results of *SyncFastGen* are included in Figure 1. However, *SyncFastGen* inherits the shortcoming of high KVC allocation failures from the block-allocation. It even increases the KVC allocation failures of vLLM and FastGen by 19% and 26%, respectively. This is because prompts in a group use up their assigned block and request a new block at the same time, exacerbating the KVC bottleneck. Finally, *SyncFastGen* has 24% higher JCT and 28% lower throughput than the *SyncCoupled* due to preemptions.

As a result, to deal with the problems in *SyncCoupled* and *MultiRes* (or *UnsyncCoupled*) (O1), we propose *SyncDecoupled*, which augments *SyncDecoupled* with decoupling PT and GT processing. *SyncDecoupled* maintains two separate waiting queues for PTs and GTs. The KVC allocated to a PT and a GT equals to its prompt length and predicted RL, respectively. PTs and GTs are responsible for fully utilizing the GPU and KVC, respetively. After a prompt is processed, its GT is entered into the GT waiting queue, and the GTs form time-synced groups to be scheduled. A small amount of KVC space is reserved for adding PTs at each iteration. The reserved space for Alpaca, ShareGPT and BookCorpus datasets was set to 1.2%, 3%, 5% of the total size empirically.

Resource uti. / Scheduling time	Coupled	Decoupled
Time-unsynced	high / low	high / high
Time-synced	low / low	low / high

Figure 3. Different combinations.

The performance of *SyncDecoupled* is included in Figure 1. Compared to *SyncCoupled* and *SyncFastGen*, it increases the throughput by 1.31× and 2×, and reduces the JCT by 35% and 12%, respectively. It is intriguing to see that it reaches TFS since PTs can be added to the batch in each iteration, and reduces JCT by 35.3% compared to *SyncCoupled*. Finally, we see that it only achieves 63%-79% KVC utilization due to its employment of exact-allocation.

Observation 3. Combining time-synced batching and decoupling request processing (*SyncDecoupled*) reduces scheduling time and maximizes GPU utilization in each iteration. However, the exact-allocation it employs cannot maximize KVC utilization though it avoids KVC allocation failures.

Figure 3 summarizes the relative performance on the scheduling time and both resource utilizations of different approach combinations. Time-synced batching helps reduce scheduling time and the decoupled method facilitates more fully utilizing the dual-resources simultaneously (indicated in red words). *SyncDecoupled* achieves the best performance. The results of *UnsyncDecoupled* will be presented in §4.

2.3 Response Length Prediction

We use the OPT-13B model to predict the RL [23] based on the prompt. We used the OPT-13B model from the Hugging Face repository [27] and fine-tuned it to output the RL. We used 10K requests from each trace for fine-tuning; 70% of the data for training and 30% for testing. All other remaining requests from each trace were used in our experiments. To optimize the training process resources, we employed the efficient training method LoRA [33] and trained the model for three epochs with a learning rate of 0.01. Instead of combining the RL prediction task with the request processing on the same LLM as in [23], we used a separate LLM running on another server (with four A100 GPUs) for RL prediction to avoid the interference on the request processing. For more details of the RL predictor, please refer to [23].

To avoid under-prediction, we could add a certain ratio of the predicted value as padding. Figures 4a-4c show the JCT decomposed to waiting time and processing time, the KVC utilization, and the percentage of under-provisioned requests for each padding ratio. The figure scales Alpaca's results by a factor of 10 (marked by "×10") to make them visible. As the padding ratio keeps increasing, the processing time decreases but the waiting time increases, leading to decrease in JCT first and then increase in JCT. This is because adding padding also decreases the KVC utilization and the percentage of under-provisioned requests, as shown in Figures 4b and 4c. The sweetspot padding ratios for Alpaca, ShareGPT and BookCorpus are 10%, 15% and 20%, which lead to 77.5%, 73.2%, and 69.8% prediction accuracy, respectively, in our experiments.

Figure 5a shows the percentages of the over-provisioning and under-provisioning KVC space over the allocated KVC space per request after adding the sweetspot padding. The over-provisioning percentages of Alpaca, ShareGPT, and BookCorpus are 2.62%, 5.62% and 5.12%, respectively, whereas the under-provisioning percentages are 9.30%, 13.42%, and 21.92%, respectively. In *SyncDecoupled*, when a request does not receive sufficient KVC due to under-prediction, there can be three solutions. First, it is preempted, and its KV data is offloaded to CPU memory using the strategy in vLLM [13]. Second, it is preempted and its KV data won't be offloaded to the CPU memory. Third, it can use the KVC originally reserved for PTs. Figure 5b shows that the percentages of preemption time over the JCT of the preempted requests for these three solutions are 12%, 4%, and 0.5%, respectively.

Observation 4. Using LLM for RL prediction can achieve 70%-78% accuracy. There exists a sweetspot padding ratio that minimizes the average JCT. Upon a KVC allocation failure, using the reserved KVC and offload-free preemption may be more efficient than the offload-based preemption.

2.4 Occupied KVC of Waiting Requests

A waiting GT occupies a certain KVC space for the tokens of its prompt and also for its previously generated tokens if it is preempted. If a prompt is chunked, it also occupies a certain KVC. In the above experiment, we also measured the occupied KVC for the queued GTs and chunked prompts. Figure 6 shows the occupied KVC space for each new GT (just transitioned from a prompt), each preempted GT and each chunked prompt. The percentages of occupied KVC in each of these categories for each dataset vary greatly. Further, those of preempted GTs are higher than new GTs.

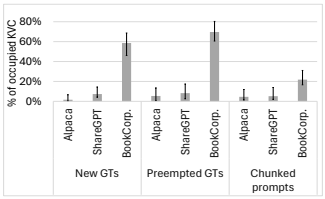
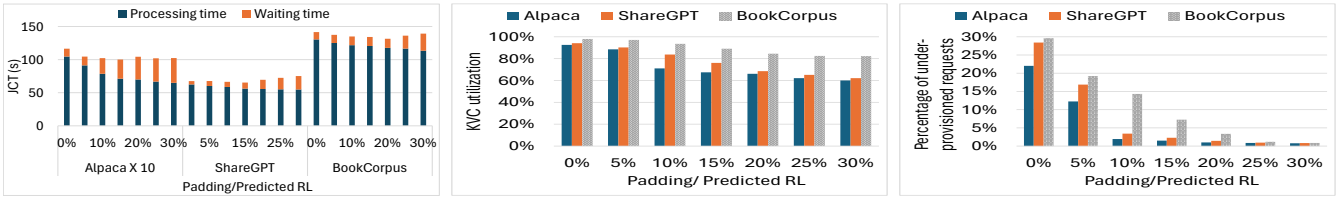


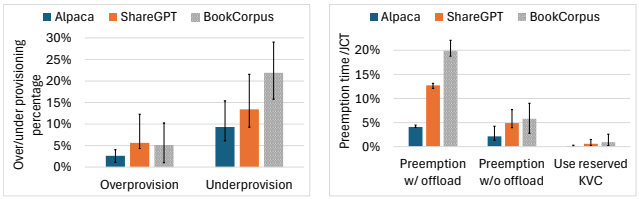
Figure 6. Occupied KVC for each queued request.

Observation 5. Waiting PTs and GTs occupy varying amounts of KVC space, so we should prioritize running those that occupy more KVC to release their KVC earlier.



(a) Response latency. **(b)** KVC utilization. **(c)** Under-provisioned requests.

Figure 4. Impact of adding padding to the predicted response length (RL).



(a) Over/under KVC allocation. **(b)** Preemption time.

Figure 5. Impact of RL misprediction.

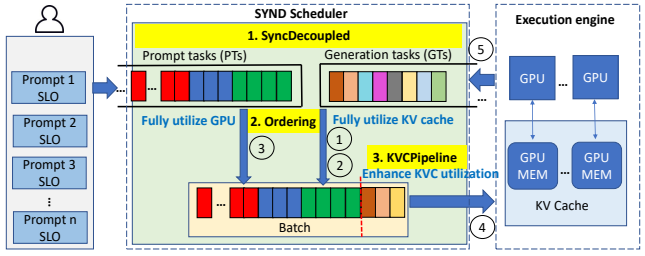


Figure 7. Architecture of EcoServe.

3 System Design of EcoServe

3.1 Overview

Based on our observations, we propose EcoServe as shown in Figure 7. It consists of the following components marked in yellow in the figure.

1. **Time-Synced Batching with Decoupled Request Processing (*SyncDecoupled*).** Guided by O2, EcoServe incorporates *SyncDecoupled*, which consists of the decoupling method (§3.2.1) and the time-synced batching method, and also handles KVC under-provisioning by using reserved KVC, padding and offload-free preemption based on O4 (§3.2.2).
2. **PT and GT Queue Ordering (*Ordering*).** It orders waiting requests, aiming to satisfy SLOs and prioritizing requests with higher occupied KVC in order to release it earlier (based on O5). It also facilitates quickly selecting requests from each queue to fully utilize each resource (§3.3).
3. **KVC Pipelining Utilization (*KVCPipeline*).** Based on O3, it enables other requests to use allocated but unused KVC space of a request to improve KVC utilization (§3.4).

In Figure 7, different colored blocks represent tokens from different requests. Users' requests are entered to the prompt waiting queue and ordered by *Ordering*. After each iteration, *SyncDecouple* is executed. Specifically, if a GT group in the batch completes, other same-RL GT groups from the GT queue are fetched to fully allocate the available KVC (①). Then, *KVCPipeline* is executed to select more GTs to use allocated but unused KVC of these selected GTs (②). Next, PTs are fetched from the prompt queue to reach the TFS to fully utilize the GPU (③). The formed batch is sent to the execution engine to be executed (④). After the execution, the PTs become GTs and are entered to the GT waiting queue and ordered by *Ordering* (⑤). Then, the process repeats again.

3.2 Time-Synced Batching with Decoupled Request Processing

3.2.1 Decoupling Prompt and GT Processing. This decoupled method aims to facilitate fully utilizing the GPU and memory resources. Users' prompts are placed in the PT waiting queue, while the GTs resulting from the prompts enter the GT waiting queue. The KVC size allocated to a PT equals its prompt length, and that allocated to a GT equals to its predicted RL. For easy memory management, the KVC allocation is still in the unit of a block [13]. In the GT queue, GTs with the same predicted RL are grouped (details are in §3.2.2). We will present how to order the PTs and GT groups in their queues in §3.3. A small amount KVC is reserved for PTs.

In the system, at the initial iteration, prompts are selected sequentially to form a batch until the available KVC space is fully allocated or the TFS is reached (i.e., GPU will be fully utilized). For subsequent iterations, the procedure is as follows. After an iteration, the PTs become GTs, which are entered into the GT queue and grouped based on predicted RL. The KV values of the prompt tokens are stored in KVC. Then, a new batch is formed for the next iteration. If there is available KVC (in the second iteration or when a GT group completes), the queued GT groups are selected sequentially until the KVC is fully allocated. If the KVC's available space cannot accommodate an entire group, the group needs to split to be accommodated into the KVC. Next, the PTs from the PT queue are selected sequentially until the TFS is reached. As a result, EcoServe can fully utilize GPU

and memory in each iteration, while reducing scheduling time.

3.2.2 Time-Synced Batching. This method enables the GTs in a group in a batch to commence and conclude execution simultaneously, eliminating the need for iteration-level scheduling. Figure 8 shows an example of RL prediction and batching same-RL GT. EcoServe groups the GTs with the same predicted RL in the GT queue. During execution, a GT group may complete earlier than other GT groups. Then, the responses of the GTs in the group are returned to the users, and other GT groups in the queue are selected to add to the batch to fully allocate the KVC.

Given the autoregressive nature of LLMs, predicting the RL of a request poses a significant challenge. Given the high perception ability of LLMs to understand input prompts and generate outputs, we use an LLM to predict the RL [23] based on the input prompt. When a request arrives at the queue, EcoServe uses the RL predictor (e.g., built upon the OPT-13B model and explained in §2) running in another server to predict its RL [23]. This RL predictor can be used for many inference servers, and it undergoes continual retraining in the system. The LLM predicts a request's RL concurrently during its waiting and PT processing. Therefore, EcoServe is suitable for the scenario where the sum of these two time latencies is no less than the RL prediction latency, which is true in our scenario with a high arrival rate. In this case, the prediction latency cost ($\approx 0.921s$ in our analysis) is unlikely to affect JCT.

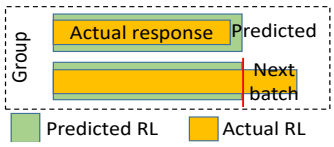


Figure 8. Batch same-RL GT.

Misprediction. There can be underestimation and overestimation. Underestimation is more critical than overestimation because insufficient KVC space causes request preemptions. In contrast, overestimation simply results in some wasted KVC space. O4 indicates that padding is an effective solution, and using reserved KVC and offload-free preemption can be effective solutions. Based on this, we propose an effective method to handle underestimation.

In the method, the predicted RL is increased by a certain percent. In addition, when a request encounters an underprovisioning, it first tries to use the reserved KVC. If the reserved KVC is not enough, this request stops processing along with other completed requests in the batch. By subtracting its currently generated response length from its predicted RL, we obtain its new predicted RL L_{new}. It will be grouped with other GTs with the predicted RLs equalling to L_{new}. For instance, as depicted in Figure 8, when the first GT (r_1) and the second GT (r_2) reach their predicted RL, r_1 completes but r_2 does not, and both GTs are returned. Subsequently, r_2 will be grouped with other GTs that share the same predicted RL as its L_{new} to be scheduled. This request won't wait long in the queue because it occupies a certain KVC and will be given a higher priority in the queue ordering method, as discussed in the next subsection.

3.3 Prompt and GT Queue Ordering

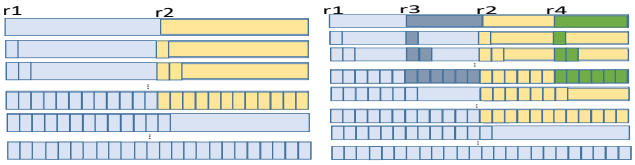
This approach determines how to order the tasks in the PT waiting queue and GT waiting queue, respectively. O5 indicates that waiting PTs and GTs occupy varying amounts of KVC space, so we should prioritize running those that occupy more KVC to release their KVC earlier.

To order the tasks in each queue, we consider three factors in order: 1) JCT SLO, 2) occupied KVC space, and 3) predicted RL for GTs and prompt length for PTs. We set this order because the SLO of a request must be satisfied, and the KVC is a bottleneck compared to GPU. In addition, in order to quickly select tasks to fully allocate KVC or utilize GPU to save scheduling time, GTs with longer predicted RLs or PTs with longer prompts should be selected earlier.

Each task has a deadline to satisfy its SLO. For each factor, we set up certain magnitude ranges and order tasks accordingly (e.g., 0.2-0.5s, 0.5-2s, >2s for the deadline, and 0-128, 128-256, 256-384, 384-512 for the predicted RL). First, we order the tasks in ascending order of the deadline. Then, within each deadline range, we further order the tasks in descending order of their occupied KVC sizes. Next, within each occupied KVC size range, we order the tasks in descending order of the predicted RL for GTs or prompt length for PTs. Consequently, when selecting GTs or PTs, EcoServe picks up the tasks in sequence and uses the binary search to find a task with the predicted RL or prompt length close to the required length to fully allocate the KVC or fully utilize the GPU.

3.4 KVC Pipelining Utilization

Motivated by O3 that the exact-allocation cannot fully utilize KVC though it prevents KVC allocation failures, we propose this *KVCPipeline* method, which allows GTs to share allocated but unused cache space. In this method, the second half of the allocated KVC part of one GT request (denoted as r_1) is reallocated to another GT (denoted as r_2). A request whose RL is less than but closest to half of r_1's RL is identified as r_2. Then, when r_1's KVC space usage reaches the middle of its allocated KVC space, i.e., the starting point of r_2's KVC space, r_2 completes and releases its occupied KVC. Here, we call r_1 the hosting GT of r_2 and call r_2 the hosted GT of r_1. Figure 9a shows an example in which different colors represent different GTs. If r_1's predicted RL (RL for simplicity) is 32 tokens, and r_2's RL is 16 tokens. Then, we allocate a 32-token KVC to r_1 and re-allocate the 16-token KVC to r_2 from the middle of the allocated space. Both GTs start running simultaneously, and when r_1's occupied KVC reaches the middle, r_2 completes and releases its cache space. Then, r_1 can continue using the cache until it completes.



(a) Example for two GTs. **(b)** Example for four GTs.

Figure 9. Examples of KVC pipelining.

We generalize this method to multiple GTs as shown in Figure 9b. Within the left and right half of the allocated space for r_1, in the second half of the space, we can further embed another GT, such as r_3 and r_4 in the figure. r_3 and r_4 will complete when their hosting GTs' KVC usages reach their memory starting points. Similarly, each of these GTs, r_1-r_4, can host another GT in itself. This process continues, akin to Russian nesting dolls, until no more GTs can be accommodated.

Recall that the RL may be under-estimated, and then a hosted GT may not complete by the predicted time. To avoid this case, we use a buffer of b tokens here; that is, EcoServe finds a hosted GT that has RL less than but closest to half of the hosing GT's RL subtracting b tokens. With the buffer strategy, if a hosted GT r_2 still does not complete when its hosting GTs need the KVC space to be returned, r_2 is preempted and its occupied cache will be moved to the main memory temporarily using the copy-on-write mechanism in [13]. Therefore, EcoServe chooses the GTs with loose SLOs in selecting hosted GTs.

In EcoServe, after the GT groups are selected in batching, for each of the GT groups with m GTs and RL $L_r = l$ in the GT waiting queue, EcoServe finds GT groups with RLs less than but closest to $l/2 - b$, $l/4 - b$, $l/8 - b$..., respectively. EcoServe needs to find $m \cdot 2^1$ GTs, $m \cdot 2^2$ GTs, $m \cdot 2^3$ GTs and so on for each of the above RL values, respectively. The GTs in the hosted GT group and in the hosting GT group are randomly mapped.

3.5 Putting All Together

Algorithms 1 shows the pseudocode of EcoServe incorporating the three methods. We list the steps in the algorithm corresponding to those in Figure 7. When a batch is returned, if a GT group completes, then EcoServe fetches GT groups from the GT queue to add to the batch. Specifically, it first selects GT groups to fully allocate the KVC (line 2), and then uses the KVC pipelining method to select the hosted GTs for the selected GTs recursively (line 3). Next, it selects the PTs from the PT queue to reach the TFS (line 5). Finally, the batch is executed (line 6). After the execution, the GTs generated from the prompts and preempted GTs are inserted into the GT queue (lines 7-9).

Algorithm 1: Pseudocode of an iteration's execution.

Input : Returned batch; Sorted PT queue (Q_P); Sorted GT queue (Q_G); available KVC (A_{KVC})
Output : Form and execute a new batch

1 **if** *a GT group completes* **then**
2 Select GT groups from Q_G until $\sum L_r = A_{KVC}$ ① (*SyncDecouple*);
3 Recursively select hosted GT groups from Q_G for each hosting GT group ② (*KVCPipeline*);
4 **end if**
5 Select PTs from Q_P until $\sum L_p = TFS$ ③ (*SyncDecouple*);
6 Execute the batch ④
/* Enter the tokens generated from PTs to GT queue */
7 **for** *each newly generated GT and preempted GT t* **do**
8 Ordering(t, Q_G); ⑤ (*Ordering*);
9 **end for**
10 Return tokens generated from GTs to users;
11 Release KVC space of completed requests;

4 Performance Evaluation

Experiment Settings. The experiment settings are the same as in §2 unless otherwise specified. We additionally tested Llama-33B and OPT-175B [27] and using model parallelism on two and eight GPUs of the AWS machine mentioned in §2, respectively, following strategies in [13]. The KVC memory for them is 19.2GB and 264GB, respectively. We set the batch size of Orca to 16 for OPT-175B as in [11]. We used the sweetspot padding ratios identified in §2 and empirically set the buffer b for *KVCPipeline* (§3.4) to 15%, 15%, and 10% of the predicted RL for Alpaca, ShareGPT, and BookCorpus, respectively. For each dataset, we measured the average prompt processing latency, t_p, and the average token generation latency, t_g. We then set the JCT SLO of a request to *SLO-scale*$\times(t_p + t_g \times l_g)$, where l_g denotes its RL, following [34]. The default SLO-scale was set to 2. We used 3-hour trace for OPT-13B and Llama-33B and used 1-hour trace for OPT-175B model.

Implementation. We implemented EcoServe using the vLLM source code [13], comprising about 7K lines of Python and 2K lines of C++ code. The KVC pipelining method was written in C++, while the batching scheduler and queue management were implemented in Python. To order the prompts and same-RL GTs, we maintained multiple priority queues from the class *queue.PriorityQueue*. The memory management process was built using the *append_token* function scheme in [13]. Required metadata, such as a table containing the start and end memory addresses of the requests, was stored in the host memory. The kernel in vLLM was adopted to read and write in the KVC. To ensure parallel memory access, a GPU thread block was assigned to read/write the memory. Allocation for the predicted RL for the request groups was handled using a different thread block. The *cudaMemcpyAsync* API was used to manage data movements between various levels of memory. Prompt and token generation were batched in one iteration as in nanoGPT [35].

EcoServe also allows parallelization of the attention computation using the *flash_attn_qkv_packed* function from the FlashAttention [36] API.

Compared Methods. We compared EcoServe with Orca, vLLM and FastGen. We also tested the variants of EcoServe. EcoServe-D (i.e., *UnsynedDecoupled*) decouples prompt and GT processing, selects tasks sequentially from the two queues to fully utilize the GPU and KVC in each iteration, and uses the exact-allocation. EcoServe-SD is *SyncDecouple*. EcoServe-SDO further incorporates the *Ordering* method. Note that the performance difference between EcoServe and EcoServe-SDO indicates the effectiveness of *KVCPipeline*. We also tested EcoServe with full knowledge of the RLs and denoted it by *Oracle*.

Normalized latency of the system is the mean of every request's end-to-end latency divided by its output length, as defined in [11, 13]. Figures 10a-10i show the normalized latency of the systems versus the request rate. A high-throughput serving system should retain low normalized latency against high request rates. We now compare the request rates that the methods can sustain while maintaining similar latencies. On ShareGPT, EcoServe can sustain 2.5-4× higher request rates compared to vLLM, 22-64× compared to Orca, and 1.25-2.25× compared to FastGen. On BookCorpus, EcoServe can sustain 2.5-2.8× higher request rates compared to vLLM, 12.5-13× compared to Orca, and 1.88-2.33× compared to FastGen. On Alpaca, EcoServe can sustain 1.13-2.14× higher request rates compared to vLLM, 5.6-9× compared to Orca, and 1.2-1.24× compared to FastGen. EcoServe's advantage on Alpaca is less pronounced because it contains short sequences, allowing for more requests to be batched and processed quickly.

For OPT-13B, EcoServe can sustain 1.20-2.33× higher request rates compared to FastGen, and 1.28-2.33× and 3.5-25× compared to vLLM and Orca, respectively. For OPT-175B, EcoServe can sustain 1.20-2.25× higher request rates compared to FastGen, and 1.28-2× and 2.8-22× compared to vLLM and Orca, respectively. Finally, for Llama, EcoServe can sustain 1.2-2.18× higher request rates than FastGen, and 2.4-4× and 3-64× compared to vLLM and Orca, respectively. EcoServe outperforms other methods because it can more fully utilize KVC and GPU, avoid KVC allocation failures, and reduce scheduling time. EcoServe's time-synced batching helps reduce scheduling time and its decoupling method helps fully utilize the dual-resources. Further, its exact-allocation and padding strategy, KVC reservation and offload-free preemption help avoid KVC allocation failures or mitigate its adverse impact, while its KVC pipelining method helps improve KVC utilization. Moreover, its ordering method helps release occupied KVC earlier, expedite finding requests to fully utilize the dual-resources and satisfy SLO requirements. in contrast, vLLM does not aim to fully utilize GPU while FastGen does not aim to fully utilize KVC, and both of them generate many KVC allocation failures due to block-allocation.

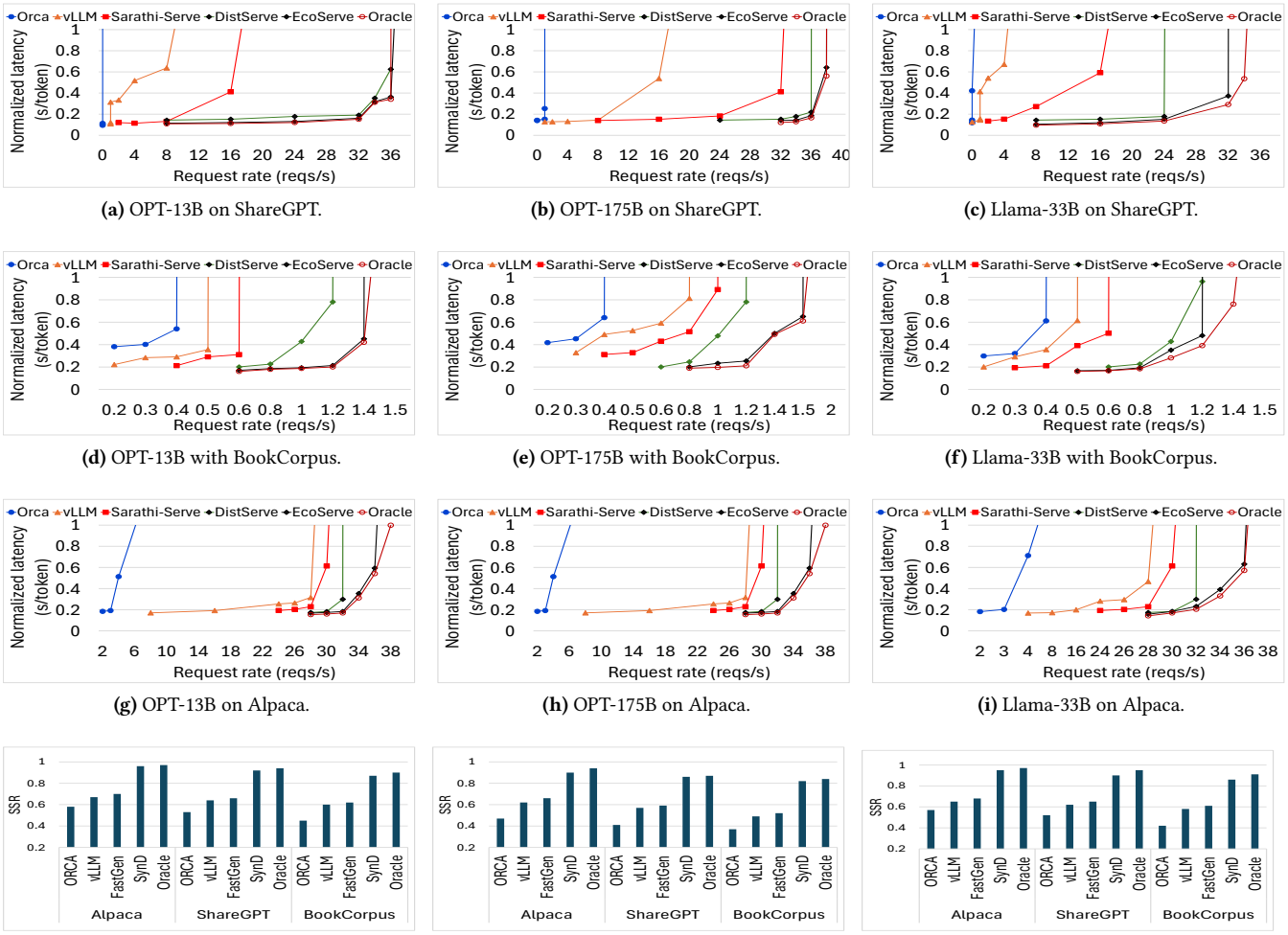
EcoServe improves Orca by a large margin because Orca has a fixed small batch size and uses max-allocation, thus limiting GPU utilization. Oracle reaches only an 8%, 11%, and 5% higher request rate than EcoServe for OPT-13B, OPT-175B, and Llama, respectively, on average across the three datasets.

Figures 10j-10l show the SLO satisfaction ratio (SSR) for each model for the three datasets, respectively. Compared to FastGen, vLLM, and Orca, EcoServe has 42%, 83%, and 93% higher SSR on average across the datasets in OPT-13B, 47%, 77%, and 1.06× higher SSR in OPT-175B, and has 41%, 80%, and 91% higher SSR in Llama. EcoServe has only 2%, 2.8%, and 4% lower SSR for the three models compared to Oracle.

Ablation Study. Figures 11a-11i illustrate the performance of individual components of EcoServe in terms of average JCT, Time Between Tokens (TBT), SSR and throughput. In the figure, "/10" means that the figure divides the results of the method by 10 to make the figure visible.

Compared to vLLM, EcoServe-D, EcoServe-SD, EcoServe-SDO, and EcoServe achieve 16-36%, 35-58%, 43-64%, and 64-91% lower JCT, 14-32%, 30-52%, 41-62%, and 60-72% lower TBT, 10-38%, 17-52%, 23-74%, and 27-91% higher SSR, and 75-84%, 91%-1.13×, 1.15-1.32×, and 1.67-1.96× higher throughput, respectively. On average, across all datasets and all models, EcoServe-D, EcoServe-SD, EcoServe-SDO, and EcoServe achieve 28%, 47%, 54%, and 83% lower JCT than vLLM, respectively. This indicates that Decoupling, Synced batching, Ordering, and KVCPipeline reduce JCT by 28%, 19%, 7%, and 29%, respectively. Additionally, EcoServe-D, EcoServe-SD, EcoServe-SDO, and EcoServe achieve 27%, 44%, 49%, and 70% lower TBT than vLLM, respectively. This suggests that the individual methods reduce TBT by 27%, 17%, 5%, and 21%, respectively. EcoServe has an average TBT of 0.141s with 5th and 95th percentile values of 0.129s and 0.221s. This indicates that GT queuing won't greatly increase their TBTs. Furthermore, EcoServe-D, EcoServe-SD, EcoServe-SDO, and EcoServe achieve 26%, 42%, 63%, and 80% higher SSR than vLLM, respectively. This implies that the individual methods increase SSR by 26%, 16%, 21%, and 17%, respectively. EcoServe-D, EcoServe-SD, EcoServe-SDO, and EcoServe achieve 78%, 1.04×, 1.21×, and 1.83× higher throughput than vLLM, respectively. This implies that the individual methods increase throughput by 78%, 36%, 17%, and 62%, respectively. Ordering is less effective at improving JCT and TBT compared to other methods, but it improves SSR by considering SLO in queue ordering.

Scheduling Time Overhead. Figures 11j-11l show the total scheduling time overhead of different methods. EcoServe have 2.93% higher scheduling time overhead than vLLM. With this slightly higher time overhead, EcoServe achieves much higher performance in other metrics than vLLM, as explained above. The scheduling overhead of the components



(a) OPT-13B on ShareGPT. **(b)** OPT-175B on ShareGPT. **(c)** Llama-33B on ShareGPT.

(d) OPT-13B with BookCorpus. **(e)** OPT-175B with BookCorpus. **(f)** Llama-33B with BookCorpus.

(g) OPT-13B on Alpaca. **(h)** OPT-175B on Alpaca. **(i)** Llama-33B on Alpaca.

(j) SSR of OPT-13B. **(k)** SSR of OPT-175B. **(l)** SSR of Llama-33B.

Figure 10. Overall performance comparison with varied request rates for all models on the three traces.

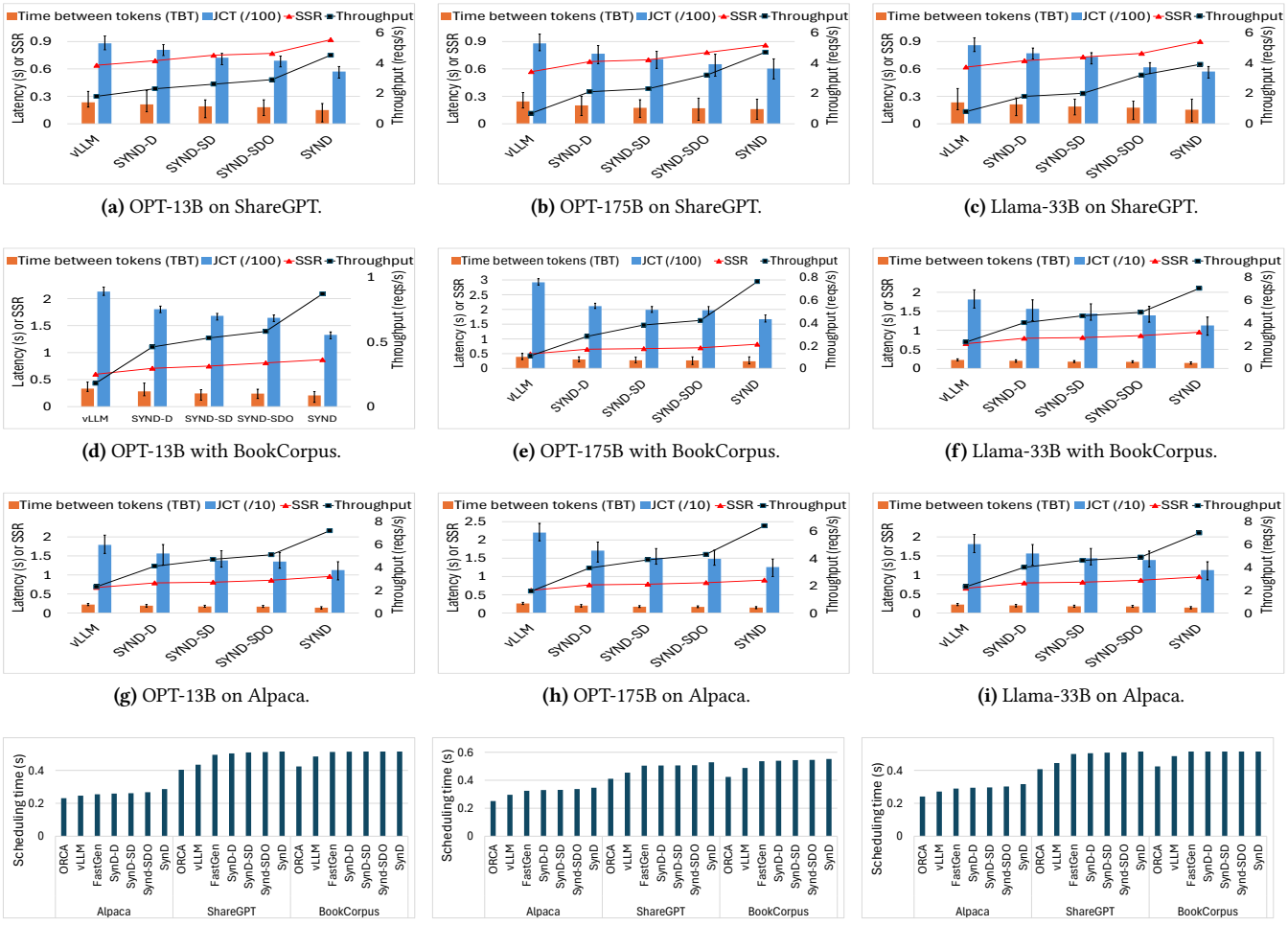
of the EcoServe contributes to its overhead. Specifically, EcoServe-D, EcoServe-SD, and EcoServe-SDO have 2.05%, 2.16%, and 2.36% higher scheduling overhead than vLLM. Orca has 2.15% lower and FastGen has 1.84% higher scheduling overhead than vLLM.

Sensitivity Testing. Figure 12 shows the normalized performance of JCT, throughput (TP), and SSR by the maximum performance value with the different settings of each influencing factor for OPT-13B. The results for the other two models show similar patterns and will be presented in the appendix after paper acceptance. Figure 12a shows as the SLO scale increases from 0.5 to 2.5, the JCT and throughput are not greatly affected, but SSR increases by 23% on average for the three datasets. This is because the SLO-scale only affects EcoServe's ordering method, which considers SLOs in queue ordering; looser SLOs increase SSRs. BookCorpus has lower SSR than other datasets due to its longer sequence lengths and longer JCTs. Figure 12b shows that Alpaca, ShareGPT, and BookCorpus have the best performance when the padding ratio equals to 10%, 15%, and 20%, respectively. The JCT follows the same trend as in Figure 4a due to the same reasons explained. Accordingly, SSR and throughput first increase and then decrease.

Figure 12c shows that the best reserved KVC percentage for Alpaca, ShareGPT and BookCorpus is 2%, 3%, and 4%, respectively. The general trend of the datasets on reserved KVC is similar to that of the padding ratio for the same reasons. Figure 12d demonstrates that as the buffer percentage increases, the performance improves initially but then decreases. The best buffer percentage for Alpaca, ShareGPT and BookCorpus is 15%, 15%, and 10%.

5 Related Work

Initially, LLM inference utilizes request-level scheduling, where all requests within a batch are processed collectively until completion [16, 17]. To overcome its drawbacks of long



(a) OPT-13B on ShareGPT. **(b)** OPT-175B on ShareGPT. **(c)** Llama-33B on ShareGPT.

(d) OPT-13B with BookCorpus. **(e)** OPT-175B with BookCorpus. **(f)** Llama-33B with BookCorpus.

(g) OPT-13B on Alpaca. **(h)** OPT-175B on Alpaca. **(i)** Llama-33B on Alpaca.

(j) Scheduling overhead of OPT-13B. **(k)** Scheduling overhead of OPT-175B. **(l)** Scheduling overhead of Llama-33B.

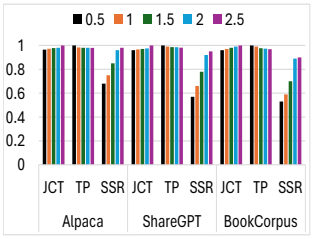
Figure 11. Ablation study of EcoServe for all models on the three traces.

response time, Orca [11] uses iteration-level scheduling and max-allocation but it generates GPU under-utilization. To address this, vLLM [13] uses a block-based approach. Jin *et al.* [19] proposed S^3, which predicts the output length and allocates the required memories to the request. Zheng *et al.* [23] proposed a system that predicts the output length of an input and then batches queries with similar response lengths. Wu *et al.* [12] proposed FastServe, employing preemptive scheduling to minimize JCT. Sheng*et al.* [37] considered achieving fairness in scheduling. Compared to existing schedulers, EcoServe distinguishes itself by its time-synced batching with decoupled request processing and KVC pipelining to fully utilize the dual resources at each iteration and avoid KVC allocation failures.

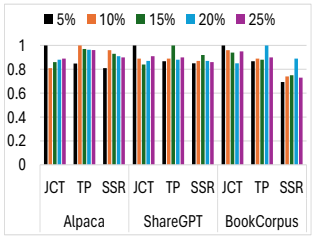
Several other approaches aim to enhance LLM performance. Sheng *et al.* [38] introduced FlexGen for memory hierarchy optimization using linear programming. Oh *et al.* [39] proposed *ExeGPT*, which finds the optimal execution configuration, including batch sizes and partial tensor parallelism, to maximize inference throughput while meeting latency constraints. Zheng *et al.* [40] proposed storing KVC in a radix tree for multiple requests. Liu *et al.* [41] proposed storing pivotal tokens based on attention scores in KVC. Lee *et al.* [42] proposed prefetching only the essential KV cache entries for computing the subsequent attention layer.

6 Conclusion

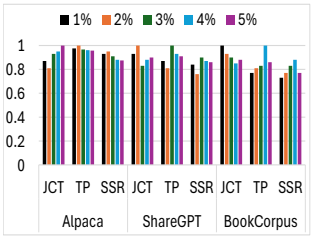
To improve throughput and latency performance of LLM inference, leveraging our findings from trace-based experimental analysis, we propose the EcoServe system, employing three key methods: 1) *SyncDecouple*, 2) *Ordering*, and 3) *KVCPipeline*. Our trace-based real experiments demonstrate that EcoServe exhibits superior performance in comparison with the state-of-the-art. We will investigate enhancing the accuracy of RL prediction and reducing prediction overhead. We also plan to investigate whether EcoServe would lead



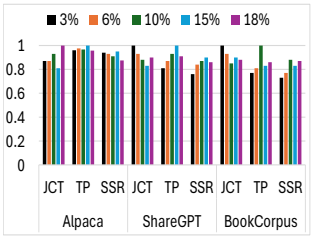
(a) SLO scale.



(b) Padding ratio.



(c) Reserved KVC.



(d) Buffer size/KVC size.

Figure 12. Effect of factors for OPT-13B on the three traces.

to imbalances in processing speeds of PTs and GTs, and how it affects the performance.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019.

[2] Ben Mann, N Ryder, M Subbiah, J Kaplan, P Dhariwal, A Neelakantan, P Shyam, G Sastry, A Askell, S Agarwal, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 1, 2020.

[3] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[4] Raisa Islam and Owana Marzia Moushi. Gpt-4o: The cutting-edge advancement in multimodal llm. *Authorea Preprints*, 2024.

[5] https://www.llama.com/, 2024.

[6] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[7] Hong Zhang 0025, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. Shepherd: Serving dnns in the wild. In *20th USENIX Symp. Networked Syst. Des. Implement. NSDI 2023 Boston MA April 17-19 2023*, NSDI 2023, pages 787–808. USENIX Association.

[8] Amazon AWS. Inference dominates ml infrastructure cost. https://aws.amazon.com/machine-learning/inferentia/, 2023.

[9] How much does it cost to run chatgpt per day?, 2024. https://www.govtech.com/question-of-the-day/how-much-does-it-cost-to-run-chatgpt-per-day: :text=According

[10] Forbes. Generative ai breaks the data center. https://www.forbes.com/sites/tiriasresearch/2023/05/12/generative-ai-breaks-the-data-center-data-center-infrastructure-and-operating-costs-projected-to-increase-to-over-76-billion-by-2028/, 2023.

[11] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *Proc. of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[12] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *ArXiv*, abs/2305.05920, 2023.

[13] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Haotong Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proc. of the 29th Symposium on Operating Systems Principles*, 2023.

[14] Microsoft. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. https://github.com/microsoft/DeepSpeed/tree/master/blogs/deepspeed-fastgen. Online; accessed in June 2024.

[15] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve, 2024.

[16] Jiacong Fang, Qiong Liu, and Jingzheng Li. A deployment scheme of yolov5 with inference optimizations based on the triton inference server. In *2021 IEEE 6th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, pages 441–445. IEEE, 2021.

[17] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[18] Nvidia. FasterTransformer. https://github.com/NVIDIA/FasterTransformer. Online; accessed in April 2024.

[19] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. S3: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, 36:18015–18027, 2023.

[20] Shweta Jain and Saurabh Jain. Analysis of multi level feedback queue scheduling using markov chain model with data model approach. *International Journal of Advanced Networking and Applications*, 7:2915–2924, 06 2016.

[21] Nvidia. Megatron-LM. https://github.com/NVIDIA/Megatron-LM. Online; accessed in April 2024.

[22] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using Megatron-LM. In *Proc.of the International Conference for High Performance Computing*, 2021.

[23] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. Response length perception and sequence scheduling: An LLM-empowered LLM inference pipeline. In *Proc. of the 37th Conference on Neural Information Processing Systems*, 2023.

[24] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.

[25] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[26] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference

time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

[27] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. *ArXiv*, abs/2205.01068, 2022.

[28] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford Alpaca: An instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[29] ShareGPT. sharegpt-english. https://huggingface.co/datasets/theblackcat102/sharegpt-english, 2023.

[30] Sosuke Kobayashi. Homemade bookcorpus. https://github.com/soskek/bookcorpus, 2018.

[31] Jongwook Choi. gpustat. https://github.com/wookayin/gpustat. Online; accessed in April 2024.

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

[33] J. Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *Proc. of the ICLR*, 2022.

[34] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing. In *Proc. of the 2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[35] NanoGPT. Karpathy. https://github.com/karpathy/nanoGPT, 2023.

[36] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *ArXiv*, abs/2307.08691, 2023.

[37] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models. In *Proceedings of OSDI*, 2024.

[38] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: high-throughput generative inference of large language models with a single gpu. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

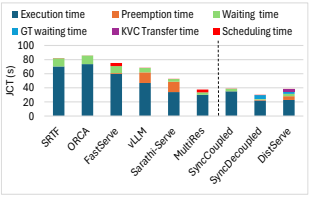
[39] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. Exegpt: Constraint-aware resource scheduling for llm inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024.

[40] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023.

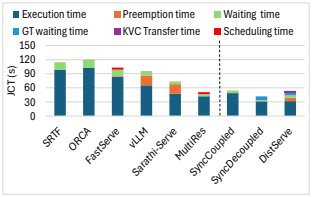
[41] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. In *Proc. of the 37th Conference on Neural Information Processing Systems*, 2023.

[42] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. Infinigen: Efficient generative inference of large language models with dynamic kv cache management. In *Proceedings of OSDI*, 2024.

A Results of ShareGPT and BookCorpus for different Schedulers



(a) JCT for ShareGPT.



(b) JCT for BookCorpus.

Figure 13. Comparison of different schedulers.

Figure 13 shows similar performance for the ShareGPT and BookCorpus as in Figure 1 (§2).