THE EFFECT OF SCHEDULING AND PREEMPTION ON THE EFFICIENCY OF LLM INFERENCE SERVING

Kyoungmin Kim [1] **Kijae Hong** [2] **Caglar Gulcehre** [1] **Anastasia Ailamaki** [1]

ABSTRACT

The growing usage of Large Language Models (LLMs) highlights the demands and challenges in scalable LLM inference systems, affecting deployment and development processes. On the deployment side, there is a lack of comprehensive analysis on the conditions under which a particular scheduler performs better or worse, with performance varying substantially across different schedulers, hardware, models, and workloads. Manually testing each configuration on GPUs can be prohibitively expensive. On the development side, unpredictable performance and unknown upper limits can lead to inconclusive trial-and-error processes, consuming resources on ideas that end up ineffective. To address these challenges, we introduce INFERMAX, an analytical framework that uses inference cost models to compare various schedulers, including an optimal scheduler formulated as a constraint satisfaction problem (CSP) to establish an upper bound on performance. Our framework offers in-depth analysis and raises essential questions, challenging assumptions and exploring opportunities for more efficient scheduling. Notably, our findings indicate that preempting requests can reduce GPU costs by 30% compared to avoiding preemptions at all. We believe our methods and insights will facilitate the cost-effective deployment and development of scalable, efficient inference systems and pave the way for cost-based scheduling.

1 INTRODUCTION

The growing demand for large language model (LLM) inference highlights challenges in efficient and scalable LLM inference systems. Unlike training, which is a one-time cost, inference is a continuous and far more expensive operation due to the high costs of GPU time and energy consumption. For example, ChatGPT approximately receives 600M visits per month[*], spending $0.7M a day to run GPUs[†]. The recent OpenAI o1 model also underscores the increased demands of LLM inference, as the model runs multiple inference paths to tackle complex reasoning tasks. Therefore, reducing the LLM inference latency leads to significant economic and environmental savings considering CO2 emissions in running models (Patel et al., 2024b; Luccioni et al., 2023).

Researchers and developers have focused heavily on creating more efficient LLM inference systems and techniques (Yu et al., 2022; Kwon et al., 2023; Agrawal et al., 2023; Zhong et al., 2024b; Lee et al., 2024; Zhu et al., 2024; Xu et al., 2024; Pan et al., 2024; Dao et al., 2022), but they have largely overlooked *scheduling* inference requests. A recent work (Agrawal et al., 2024b) shows that mis-scheduling can degrade throughput by more than six times. However, the optimal scheduling policy is yet to be known. Instead of naively repeating trials and errors in developing more efficient schedulers, we need a more systematic and cost-effective approach. Figure 1 illustrates this motivation, that taking a glance at the optimal objective would improve the productivity of development and save costs substantially.

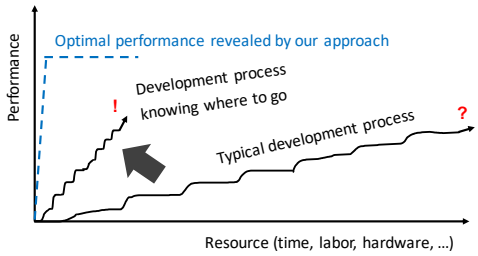


Figure 1. An example that illustrates the need for understanding the achievable limits when developing LLM inference systems and techniques. One of our primary goals is to estimate optimal performance without incurring high hardware and development costs.

It is also non-trivial to predict how the optimal scheduler would behave, since it actually chooses counter-intuitive

[1]École Polytechnique Fédérale de Lausanne, Switzerland [2]CERES TECHNOLOGIES, South Korea. Correspondence to: Kyoungmin Kim <kyoung-min.kim@epfl.ch>.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2024.

[*]https://explodingtopics.com/blog/chatgpt-users

[†]https://seo.ai/blog/how-many-users-does-chatgpt-have, https://www.semianalysis.com/p/the-inference-cost-of-search-disruption

policies which we find out in this paper. For example, multiple requests face a race condition as they store their intermediate data (KV, Section 2.1) in the limited GPU memory. If memory runs out, the system should preempt and restart some requests later, so a common belief is avoiding preemption and recomputation overheads. However, one of our findings reveals that the optimal scheduler makes the choice to preempt and restart requests, particularly when the requests are short and memory is largely utilized. This can save more than 30% of GPU time compared to waiting for other requests to release memory to avoid any preemptions. Preempting long requests can rather degrade performance by 30% due to longer recomputation times. Such a correlation between request length and preemption is challenging to uncover.

To tackle the challenges, we propose INFERMAX, an analytical framework to compare different schedulers and assess scheduling policies against the optimal ones. INFERMAX extends VIDUR (Agrawal et al., 2024a), a framework that allows simulation of schedulers on diverse hardware and model configurations. This offers cost-effectiveness as the simulation does not require running GPUs once the inference costs are modeled. While VIDUR focuses on the simulation and searching for the optimal hardware-model-scheduler configuration for deployment, it lacks a thorough analysis and exploration of further opportunities to enhance performance, leaving a gap for development. We formulate the problem of finding optimal schedules for the first time using the constraint satisfaction problem (CSP) (Schrijver, 1998) based on the cost models. Here, one can force particular scheduling policies in forms of constraints and optimize latency, throughput, or any objective that can be represented as a formula.

Our analysis using INFERMAX measures the effect of scheduling on inference performance, revisiting the overlooked scheduling policies and using the CSP solutions as the optimal policies to pursue. Our key findings include the correlation between request length and preemption. From the analysis, we envision a *cost-based* scheduling as the cost-based query optimization in databases (Selinger et al., 1979), which generalizes across a variety of inference systems with unique policies (e.g., KV cache offloading (Lee et al., 2024; Pan et al., 2024) and just-in-time KV compression) considering diverse storage hierarchies.

In summary, we propose an analytical framework INFERMAX for analyzing schedulers (Section 3), formulate the problem of finding optimal schedules as CSP, offering theoretical upper bound on the performance (Section 3.3), and provide a comprehensive analysis on schedulers that worth 200 GPU hours, showing that 30% of latency can further be saved by harnessing preemptions (Section 4). Finally, we discuss our vision (Section 5).

2 BACKGROUND

This section explains the processing of LLM requests and related work.

2.1 Processing a Single LLM Request

In LLM inference, the input text submitted by a user is called the *prompt*. When the system receives a prompt, it processes its tokens to generate a new token, a phase known as *prefill*. This is followed by *decode* steps, where each step generates a new token based on the last generated one. Each step involves (1) embedding the input tokens into vector representations, (2) transferring these vectors to the GPU, and (3) executing matrix multiplications between input vectors and model weights, along with other GPU operations like *attentions* (Vaswani et al., 2017). The system repeats the step (3) across all model layers, producing output vectors that match the size of the input vectors. It then feeds the final layer's output to a sampling component to generate the next token.

Most LLMs rely on the Transformer architecture, which computes the attention output for each token by using its query (Q) along with the key (K) and value (V) vectors of previous tokens. To reduce recomputation, key-value (KV) caching (Dai et al., 2019) stores previously computed KVs in GPU memory, allowing the system to reuse them for future tokens. This is widely adopted for efficient LLM inference. In this structure, prefill tokens lack previous tokens for KVs, making their processing *compute-bound*. In contrast, decode tokens have an expanding set of KVs to read as output length grows, causing decode steps to become increasingly *memory-bound* (Agrawal et al., 2023).

2.2 Processing Multiple LLM Requests

LLM inference systems rely on schedulers to batch requests at each step, prioritizing resource utilization, particularly the maximum KV cache size, M, which corresponds to the token limit. This is typically calculated as: $\frac{available\_GPU\_memory - model\_size}{KV\_size\_per\_token}$ with available GPU memory often capped at a percentage of total capacity, such as 90% (Kwon et al., 2023). The scheduler also designates C, the maximum number of tokens to process per batch.

To improve inference efficiency, systems adapt techniques from databases and operating systems. Two standard methods include continuous batching (Yu et al., 2022) and paged attention (Kwon et al., 2023), each optimizing computational and memory efficiency. Continuous batching schedules new requests step-by-step as resources become available, avoiding idle waiting times, while paged attention enhances memory utilization by managing KV caches in page segments rather than reserving large allocations per request.

Other methods, such as hybrid batching and chunked prefill (Agrawal et al., 2023; 2024b), offer alternative scheduling for LLM requests. Hybrid batching processes both prefill- and decode-phase requests together, though prefills, due to large prompts, often consume more resources and can delay decode-phase requests in the same batch. Chunked prefill mitigates this delay by enabling partial prompt processing, with a parameter P (where $P \leq C$) indicating the maximum number of prefill tokens per batch. A new token can only be generated once all prompt tokens are processed.

To further manage scheduling, we define two metrics: batch size, representing the number of requests in each batch, and running size, indicating the count of active (running) requests holding KV caches. When the total number of KVs of running requests reaches the cache budget M, the system may need to preempt some requests, clearing their KV caches in a process we refer to as *eviction*. Once evicted, a request's generated tokens are appended to the prompt. We call processing these tokens again as the *refill* phase. Notably, KV data is recomputed, rather than offloaded to other storage and reloaded, as PCIe bandwidth between GPU and external storage is lower than the GPU's processing capacity (Kwon et al., 2023).

2.3 Related Work

LLM Inference System. ORCA (Yu et al., 2022) introduces continuous batching, VLLM (Kwon et al., 2023) proposes paged attention, and SARATHI (Agrawal et al., 2023; 2024b) implements hybrid batching and chunked prefill. Instead of hybrid batching, DISTSERVE (Zhong et al., 2024b) and DEJAVU (Strati et al., 2024) disaggregate prefill and decode phases by using different GPUs, sending the KV cache from the prefill- to decode-handling GPU. VTENSOR (Xu et al., 2024) decouples the KV cache allocation and attention computation in VLLM, removing the page address translation overhead. INFINIGEN (Lee et al., 2024) offloads the KVs to CPU memory to extend the KV cache and reloads the KVs from CPU layer-wise. To minimize the data transfer overhead, it streams only the KVs of the most important tokens and approximates the full attention result. NANOFLOW (Zhu et al., 2024) proposes a finer-grained batching than the continuous batching, where each batch of requests is further partitioned into nano-batches. INSTINFER (Pan et al., 2024) uses flash drives to offload KVs and attention computations. Our approach to inference cost modeling and optimal scheduling could adapt to these advanced systems, leveraging diverse storage hierarchies and data movement strategies for enhanced performance.

LLM Inference Simulation. VIDUR (Agrawal et al., 2024a) provides execution time data for GPU operators across various models and GPU configurations, implementing multiple schedulers, including those of VLLM and SARATHI, and achieving prediction accuracy within a 9% relative error. However, VIDUR primarily emphasizes simulation and search, lacking an in-depth analysis or a structured methodology for designing improved schedulers. In contrast, LLMVIEWER (Yuan et al., 2024) uses theoretical hardware limits, such as GPU FLOPS and memory bandwidth, to analyze the performance of single batches across different hardware setups, without addressing scheduling aspects. Our approach builds on VIDUR's results, with the potential to further integrate LLMVIEWER's hardware bounds for a more comprehensive optimization.

Output Size Prediction. Several studies aim to estimate or rank the output sizes of requests to improve scheduling efficiency (Qiu et al., 2024; Zheng et al., 2023b; Fu et al., 2024). Our approach complements this work by examining the fundamental question of how much improvement potential exists in leveraging output sizes for scheduling. As detailed in Section 4, our findings also apply to scenarios where requests have identical output sizes, and shed light on input sizes when requests have heterogeneous input and output sizes.

3 INFERMAX

This section introduces INFERMAX, our analytical framework for evaluating LLM inference performance, illustrated in Figure 2. Given an initial configuration (❶), built-in or custom schedulers generate schedules (❷), defining the set of requests processed in each batch. These schedules can then be dispatched to inference systems (❸) to execute and measure performance (❹). However, this approach requires substantial development effort to establish a unified interface, enabling output from various schedulers to be sent across different inference systems, as well as for standardizing execution results for performance evaluation. Running every schedule on GPUs also incurs high computational costs.

To address these challenges, we adopt an alternative approach in this study, predicting batch execution times based on the number of tokens processed and the KV caches accessed (❺), using results from VIDUR (❻). As an alternative, theoretical hardware bounds, such as those used in LLMVIEWER, may also be applied and visualized through roofline models (❼) (Yuan et al., 2024).

The blue boxes and solid arrows in Figure 2 represent our primary focus areas. In Section 3.1, we propose a unified algorithm for representative schedulers and their variants, examining the impact of design choices on performance in Section 4. For batch time prediction, we employ simple linear models as cost models in Section 3.2. Lastly, we define the task of finding optimal schedules as a constraint satisfaction problem (CSP) in Section 3.3.

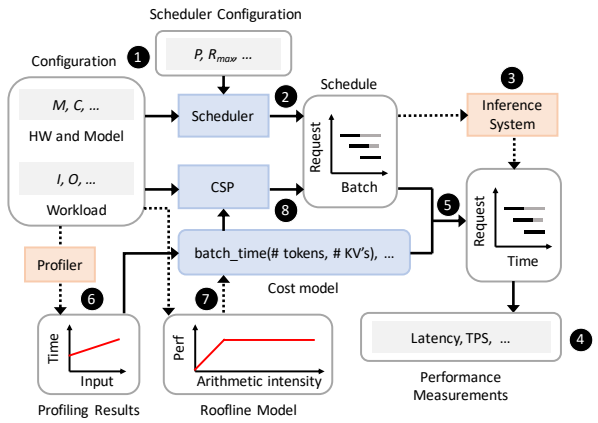


Figure 2. Overview of INFERMAX. We focus on the flows with solid arrows in the paper.

3.1 Unified Algorithm for Schedulers

This section details a unified algorithm (Algorithm 1) for schedulers across various inference systems, with distinctions in the implementation of the GETNEXTBATCH function, particularly in steps 1-4. Lines 1-7 of the algorithm are shared, as they iteratively process each batch of requests.

The requests in $\mathcal{R}_w$ and $\mathcal{R}_r$ are first ordered and grouped (Line 11, 1). For each group $\mathcal{G}$ of requests and its candidate request $cand$, if hybrid batching is disabled and $cand$ is in a different phase (prefill or decode) than the requests already in $\mathcal{B}$, $cand$ is skipped (2). In step 3, CANALLOCATE checks if $cand$ can be added to $\mathcal{B}$ based on the following conditions:

- C: Ensures the token capacity C isn't exceeded by adding $cand$, which can process up to $C - \sum_{r \in \mathcal{B}} r.c$ tokens; $r.c$ denotes the number of tokens to process for a request r.
- P: If $cand$ is in the prefill phase, it can process up to $P - \sum_{r \in \mathcal{B}} r.c$ tokens.
- M: Checks if the KV cache can store $cand$'s required KVs (one for decode, input size for prefill).
- R_{max}: Checks if the running size hasn't reached R_{max}.

For chunked prefill, $cand.c$ can be adjusted to a smaller value, $min(cand.c, P - \sum_{r \in \mathcal{B}} r.c)$, if $cand$ is in prefill phase. If allocation fails due to insufficient M, the scheduler can opt to evict other running requests and reassess $cand$ (4) or proceed to the next group. If allocation fails for other reasons, no eviction occurs. Any evicted request is removed from $\mathcal{R}_r$ and appended to $\mathcal{R}_w$. If $cand$ is successfully allocated, it is added to $\mathcal{B}$ and removed from its original queue (Line 17).

We illustrate using two representative schedulers for LLM inference: VLLM and SARATHI. In VLLM, the requests are organized into two groups, $\mathcal{R}_w$ and $\mathcal{R}_r$, with priority given to prefill requests in $\mathcal{R}_w$ (1). Hybrid batching and chunked prefill are disabled (2 and 3). If memory M is insufficient for allocation in 4, VLLM skips to the next group for candidates in $\mathcal{R}_w$. For candidates in $\mathcal{R}_r$, it attempts to evict other low-priority running requests; if this fails, it eventually evicts the candidate itself. SARATHI, on the other hand, creates three groups: $\mathcal{R}_r^d$, $\mathcal{R}_r^p$, and $\mathcal{R}_w$, where $\mathcal{R}_r^d$ and $\mathcal{R}_r^p$ represent running requests in the decode and prefill phases. This setup prioritizes decode requests (1), and both hybrid batching and chunked prefill are enabled (2 and 3). If M is insufficient for $\mathcal{R}_r^d$ candidates in 4, SARATHI attempts evictions.

Algorithm 1. SCHEDULE(M, C, P, R_{max})

Input: Max KV cache size M, max # tokens per batch C, max # prefill tokens per batch P, max running size R_{max}

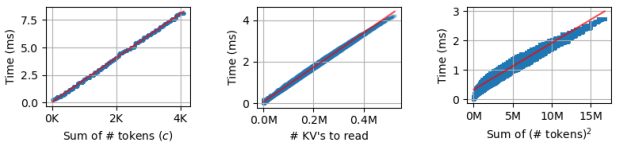
1: $\mathcal{R}_w \leftarrow \emptyset$ /* queue of waiting requests */
2: $\mathcal{R}_r \leftarrow \emptyset$ /* queue of running requests */
3: **while** (true) **do**
4: $\quad \mathcal{R}_w \leftarrow \mathcal{R}_w \cup$ GETNEWREQUESTS()
$\quad$ /* schedule batch $\mathcal{B}$ from $\mathcal{R}_w \cup \mathcal{R}_r$ */
5: $\quad \mathcal{B} \leftarrow$ GETNEXTBATCH($\mathcal{R}_w, \mathcal{R}_r, M, C, P, R_{max}$)
$\quad$ /* Process $\mathcal{B}$, send outputs to users, remove completed requests from $\mathcal{B}$ */
6: $\quad \mathcal{B}' \leftarrow$ PROCESS($\mathcal{B}$)
$\quad$ /* append remaining $\mathcal{B}'$ to $\mathcal{R}_r$ */
7: $\quad \mathcal{R}_r \leftarrow \mathcal{R}_r \cup \mathcal{B}'$

8: **Function:** GETNEXTBATCH($\mathcal{R}_w, \mathcal{R}_r, M, C, P, R_{max}$)
9: **begin**
10: $\quad \mathcal{B} \leftarrow \emptyset$ /* batch of requests */
11: $\quad$ **foreach** ($\mathcal{G} \in$ GROUPREQUESTS($\mathcal{R}_w \cup \mathcal{R}_r$) 1) **do**
12: $\quad\quad$ **foreach** ($cand \in \mathcal{G}$) **do**
13: $\quad\quad\quad$ CHECKHYBRIDBATCHING 2
14: $\quad\quad\quad$ **while** (¬CANALLOCATE($cand, \mathcal{B}$) 3) **do**
15: $\quad\quad\quad\quad$ GOTONEXTGROUPOREVICTCONTINUE 4
16: $\quad\quad\quad$ **else**
17: $\quad\quad\quad\quad \mathcal{B} \leftarrow \mathcal{B} \cup \{cand\}$,
18: $\quad\quad\quad\quad$ REMOVEREQUEST($cand, \mathcal{R}_w, \mathcal{R}_r$)
19: $\quad$ **return** $\mathcal{B}$

3.2 Cost Models for Batch Times

This section explains the prediction model used for estimating batch times. Figure 3 illustrates examples of GPU processing times for various operators within a model layer. Figure 3(a) shows the total time required for non-attention operators, such as MLPs and activations, which depend on the number of tokens processed in a batch. Figure 3(b) presents the time for decode-phase attention, determined by the number of KVs read from the cache. In contrast, prefill-attention time in Figure 3(c) scales with the square

of the number of tokens processed, $\sum_{r \in \mathcal{B}} r.c^2$. To predict batch time, we sum the costs of non-attention operators and the attention costs, using either prefill- or decode-attention based on the request phase. For hybrid batches, both prefill- and decode-attention costs are included.



(a) Non-attention. (b) Decode-attention. (c) Prefill-attention.

Figure 3. GPU times measured in (Agrawal et al., 2024a) for a layer of the Llama-2-7B model on a A100 GPU. Red lines are linear regression models with R^2 scores of 0.999, 0.997, and 0.961.

The figures demonstrate that linear regression effectively models these operator costs. This linear relationship is reasonable, as matrix multiplications and data transfers scale linearly with input vector size, and hence with the number of tokens (Zhu et al., 2024). Decode-attention, being memory-bound, is limited by KV read speed, while prefill-attention is compute-bound, exhibiting quadratic complexity (Agrawal et al., 2023). Bias terms capture additional fixed costs, such as loading model weights and initiating kernels. Improving GPU bandwidth and FLOPs, or reducing model and KV size, can lower these model coefficients and biases. Future work includes a unified cost model across hardware, model types, and systems, and refining this cost model using reinforcement learning (RL) or Bayesian optimization (Snoek et al., 2012).

With both schedulers and a batch time prediction model in place, we can now evaluate scheduler performance. Here, we primarily consider GPU kernel times as the main overhead, consistent with (Agrawal et al., 2024a). CPU overheads related to scheduling and KV cache management can be largely overlapped with GPU processing in recent inference systems, as batches are scheduled asynchronously, with GPU computation as the critical path (Zhu et al., 2024).

3.3 Optimal Scheduling as Constraint Satisfaction Problem

This section applies the constraint satisfaction problem (CSP) (Schrijver, 1998) to determine optimal schedules. Rather than seeking a better scheduling algorithm without assured performance outcomes, solving the CSP approach directs us toward optimal schedules, allowing for a more goal-oriented development process, as illustrated in Figure 1. For instance, we can validate whether a better scheduler exists that could reduce the latency of current schedulers by 10% for specific workloads. The optimization target can be adjusted to meet objectives such as latency, throughput, or fairness, if these can be represented in a formula.

We assume that the output sizes of requests are known in advance (estimating them is orthogonal to our work, Section 2.3), and that a single GPU is in use. This approach could be extended to multiple GPUs by incorporating the linear cost modeling of data transfer.

In our CSP formulation, we first establish key notations. The index $i \geq 1$ and $j \geq 1$ represent the i-th request r_i and j-th batch $\mathcal{B}_j$. We also use $j = 0$ as a virtual index to denote the initial system state. Each r_i has an input size (number of prompt tokens) I_i and output size O_i. We use I as the indicator variable, 1 if $condition$ holds or 0 otherwise. Now we explain the three parts of our CSP: variables, constraints, and objectives.

Variables. The variable $I_{i,j}$ represents the input size after processing batch $\mathcal{B}_j$, as input size may increase following evictions, with $I_{i,0} = I_i$. $m_{i,j}$ denotes $r_i.m$ after processing $\mathcal{B}_j$, with $m_{i,0} = 0$. The maximum memory usage of r_i can reach $I_i + O_i - 1$, since the last generated token doesn't need to be cached for the next token generation. $d_{i,j} = \mathbb{I}_{m_{i,j-1} \geq I_{i,j-1}}$ indicates if r_i has processed all prompt tokens before $\mathcal{B}_j$, which implies that it is in the **d**ecode phase. $g_{i,j}$ and $e_{i,j}$ indicate whether r_i **g**enerates a token or is **e**victed at $\mathcal{B}_j$. $c_{i,j}$ denotes $r_i.c$ at $\mathcal{B}_j$, while $cz_{i,j} = \mathbb{I}_{c_{i,j}>0} = \mathbb{I}_{r_i \in \mathcal{B}_j}$, $mz_{i,j} = \mathbb{I}_{m_{i,j}>0}$, and $dg_{i,j} = \mathbb{I}_{d_{i,j} \wedge g_{i,j}}$.

Constraints.

The constraints establish the interactions between variables and the conditions necessary for a valid schedule. We omit the binary variable constraints here for brevity and focus on the remaining constraints:

Termination Constraint: r_i must generate O_i tokens.

$$\forall i : \sum_j g_{i,j} = O_i \tag{1}$$

Non-Decreasing Input Size: $I_{i,j} = max(I_{i,j-1}, m_{i,j-1} + 1)$, where +1 represents the last token generated but not yet processed.

$$\forall i,j : I_{i,j} = \begin{cases} m_{i,j-1} + 1 & \text{if } e_{i,j} = d_{i,j} = 1 \\ I_{i,j-1} & \text{otherwise} \end{cases} \tag{2}$$

Memory Management: m should be zero if r_i is evicted or should increase by $c_{i,j}$. If $d_{i,j} = 0$, m must not exceed I to prevent overlapping prefill and decode phases.

$$\forall i,j : m_{i,j} = \begin{cases} 0 & \text{if } e_{i,j} = 1 \\ m_{i,j-1} + c_{i,j} & \text{if } e_{i,j} = 0 \\ \leq I_{i,j-1} & \text{if } e_{i,j} = d_{i,j} = 0 \end{cases} \tag{3}$$

Control of c in Evict/Decode Phases.

$$\forall i,j : c_{i,j} = \begin{cases} 0 & \text{if } e_{i,j} = 1 \\ \leq 1 & \text{if } d_{i,j} = 1 \end{cases} \tag{4}$$

Batch Constraints: Ensures global constraints per batch.

$$\forall j : \sum_i c_{i,j} \leq C, \sum_i c_{i,j} - dg_{i,j} \leq P, \\ \sum_i m_{i,j} \leq M, \sum_i mz_{i,j} \leq R_{max} \tag{5}$$

In this configuration, hybrid batching and chunked prefill are both enabled, as no constraints enforce separation of prefill and decode requests or require a one-time increase of $m_{i,j}$ by the prompt size. To disable these features, specific constraints can be added.

For implementing conditional constraints based on variables, we use the Big-**M** method (Pistikopoulos, 1998) to linearize them, since linear programs are more efficient than non-linear ones (Bertsimas & Tsitsiklis, 1997). For example, the upper part of (2) can be linearized as

$$I_{i,j} \leq m_{i,j-1} + 1 + \mathbf{M}(1 - e_{i,j}) + \mathbf{M}(1 - d_{i,j}) \\ I_{i,j} \geq m_{i,j-1} + 1 - \mathbf{M}(1 - e_{i,j}) - \mathbf{M}(1 - d_{i,j}) \tag{6}$$

where **M** is a sufficiently large constraint. We implement our CSP using GUROBI[‡].

Objective. The CSP objective can be set to minimize total latency, utilizing our batch time prediction model from Section 3.2. For example, $\sum_i c_{i,j}$ represents the tokens processed $\mathcal{B}_j$.

Online Setting. Supporting an online setting, where each request r_i has an arrival time T_i, is straightforward. We add variable Acc_j to track accumulated batch times and set $c_{i,j} = m_{i,j} = 0$ if $Acc_j < T_i$.

Alternative Objectives. Besides minimizing latency, we can optimize request-level metrics, such as Time-to-First-Token (TTFT) and Time-Per-Output-Token (TPOT), which are widely used. TTFT measures the time until the first output token, while TPOT measures the time between consecutive tokens. For instance, minimizing TTFT for request r_i could be achieved by using $min_j(Acc_j \cdot g_{i,j} - T_i)$ as the objective. Constraints can also set specific goals, such as verifying if a better schedule exists by running another scheduler with latency L and ensuring the new latency is under $0.9L$.

Challenge. A primary challenge in CSP is its limited scalability, as CSPs are generally NP-complete (Russell & Norvig, 2020). The complexity grows with the number of requests and batches, reaching millions of variables for 1,000 requests and 1,000 batches. Consequently, we primarily use CSP to validate findings in controlled environments (Section 4) and initialize CSP with outputs from other schedulers as seeds. Optimizing CSP for larger scales remains an area for future work.

[‡]https://www.gurobi.com

Table 1. Taxonomy of schedulers. Each scheduler also has an unlisted eviction-free variant in which no requests are evicted.

Scheduler	Priority	Hybrid Batch	Chunked Prefill	P
vLLM	Prefill	X	X	4096
SARATHI	Decode	O	O	512
SARATHI$_{P=C}$	Decode	O	O	4096
SARATHI$_{nocp}$	Decode	O	X	4096
vLLM$_{hy}$	Prefill	O	X	4096
SARATHI$_{nohy}$	Decode	X	X	4096

4 ANALYSIS

This section evaluates the performance of different schedulers and addresses key questions, some of which have received limited attention in previous studies.

4.1 Setup

Model and Hardware. We follow the default configuration of (Agrawal et al., 2024a), using the Llama-2-7B model on an A100 GPU, and set $C = 4096$ and $M = 100$K as default.

Schedulers. Using (Agrawal et al., 2024a) as a reference, we employ the vLLM (Kwon et al., 2023) and SARATHI (Agrawal et al., 2024b) schedulers as baselines. To better analyze scheduler performance, we vary R_{max} from 128 to unlimited, aiming to assess the effect of key scheduler features. Additionally, we determine the optimal R_{max} value in Section 4.6. Table 1 lists the schedulers we compare, excluding certain variants that either performed similarly or worse in our tests. We also implement eviction-free versions of the schedulers by reserving the maximum possible memory usage per candidate in the KV cache (in CANALLOCATE in Algorithm 1).

Workloads. For clarity, we begin with fixed input and output sizes I and O for all requests, ranging from 1 to 1024. We select request counts of $B = 32$ and $B = 1024$ to represent low and high contention scenarios. To simplify analysis, we use an offline setting where all requests are available before scheduling begins, which acts as a snapshot of an online environment. Analysis of the online setting is deferred to future work.

Metrics. We measure system performance using total latency and tokens-per-second (TPS), calculated as the number of generated tokens divided by latency. Average TTFT and TPOT are used to evaluate request-level performance.

4.2 Under High Contention

We focus on the high-contention scenario with $B = 1024$ as low contention with $B = 32$ does not trigger evictions, and its results are covered in subsequent sections (see Appendix A.1 for low contention details). In this analysis, we categorize the schedulers in Table 1 by performance into

groups: {SARATHI }, {SARATHI$_{P=C}$, SARATHI$_{nocp}$ }, and {vLLM, vLLM$_{hy}$ }. We exclude SARATHI$_{nohy}$ due to its significantly higher latency and TTFT (Appendix A.1). Figure 4 displays the results for SARATHI, SARATHI$_{P=C}$, and vLLM as representatives.

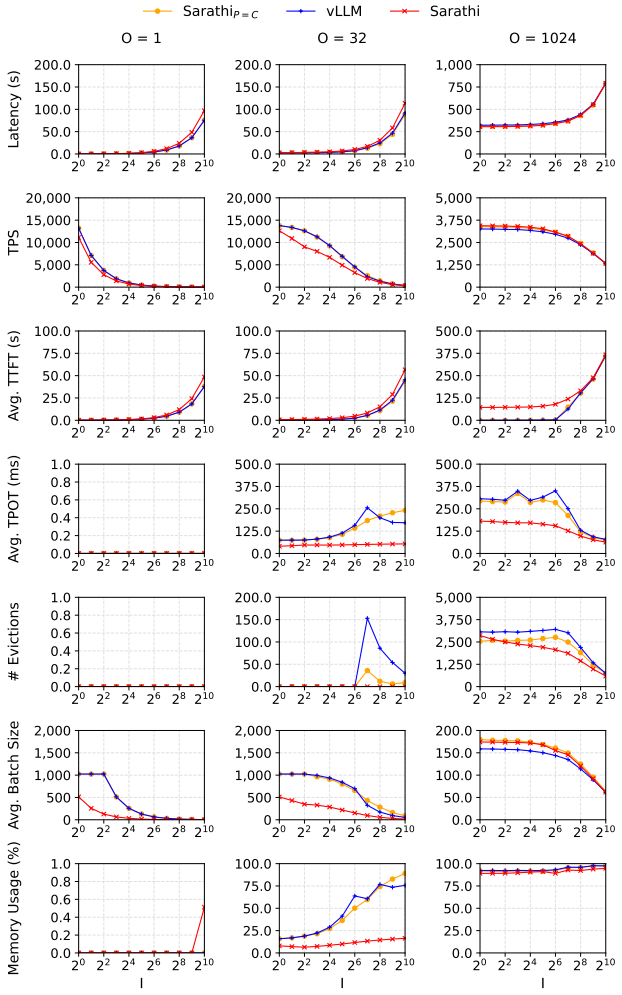


Figure 4. Results for large B of 1024. Each column represents a specific O, with I on the x-axis.

Overall, as I and O increase, latency rises across all schedulers. vLLM demonstrates the lowest latency and highest TPS by prioritizing prefill requests and batch processing decodes in parallel, resulting in large batch sizes, except when high O values lead to frequent evictions. Eviction rates increase with O because each request competes to retain its KV cache. SARATHI generally has lower TPS (up to 40.9% below vLLM) but achieves a more stable TPOT, up to 5x lower, due to balanced handling of prefill and decode phases through hybrid batching and a smaller P relative to C. SARATHI$_{P=C}$ strikes a middle ground, achieving nearly the highest TPS for each O, though it resembles vLLM more closely as it processes up to C prefill tokens per batch, matching vLLM's prefill speed by managing up to C/I new running requests per batch. Below, we discuss these trends in further detail.

TPS. TPS peaks as B grows, allowing more requests to process concurrently, generating more tokens per batch. However, TPS declines as I increases due to higher prefill costs. Beyond a certain point of O, TPS decreases because small O values make prefill dominant, while large O values make subsequent decode batches heavier, as decode costs scale linearly with the number of tokens or KVs read (Figure 5).

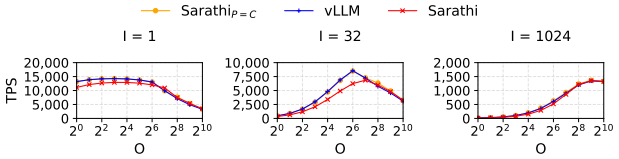


Figure 5. TPS for a large $B = 1024$.

TTFT and TPOT. TTFT and TPOT reveal a trade-off, where TTFT reflects prefill time and TPOT reflects decode time. vLLM and SARATHI$_{P=C}$ achieve lower TTFT but higher TPOT than SARATHI. Schedulers other than SARATHI can process up to C prefill tokens per batch, starting prefill early, which minimizes TTFT but increases TPOT due to larger batch sizes and more KVs read.

An interesting point is that TPOT decreases beyond a certain value of I. For large O (e.g., 1024), this is due to reduced evictions, as frequent evictions cause refills to dominate runtime. In contrast, for small O, refills have a lesser impact, so TPOT more directly reflects batch size and the number of KVs, as previously noted.

Evictions and batch size. The parameter M limits the maximum running requests in prefill or decode phases, while the number of new running requests per batch is limited by prefill tokens P. As maximum P/I requests can be newly run, high I decrease both batch size and running requests, leading to fewer evictions as I rises in Figure 4. However, I and O also determine memory reserved per request, so small values for I and O result in less frequent evictions.

The key distinction between I and O in terms of their impact on evictions is that I represents the immediate memory reserved, whereas O determines the peak memory usage after approximately $\Omega(O)$ batches have been processed. Consequently, schedulers that only consider I and disregard O risk overloading the system by batching requests without fully understanding their long-term memory demands. As O grows, this can lead to a significant increase in the number of evictions.

Memory (KV Cache) Usage. For $O = 1$, only SARATHI shows a positive KV count at $I = 1024$, as it partitions I

into smaller chunks with $P = 512 < I$, storing KVs for partially processed prompts in memory. Other schedulers complete requests in a single step by generating one token and releasing the prompt.

Because SARATHI gradually introduces running requests, it maintains stable memory consumption for KVs when $O = 32$ as new requests start, some requests complete and release their KVs. However, when $O = 1024$, SARATHI experiences high memory demand, occupying more than 90% of the KV cache.

Remark. Schedulers face a basic TTFT-TPOT trade-off. SARATHI maintains a balanced TTFT and TPOT across varying I, O, and B. Other schedulers excel with moderate values but encounter eviction spikes under high memory demands due to aggressive batching. SARATHI mitigates this with a smaller P relative to C. High TPOT results primarily from evictions, with batch size and KV load as secondary factors. Evictions rise with larger B and O values, while I acts as both a limiting factor (as new running requests are bounded by P/I and an increasing factor (by raising memory needs). Testing on a live inference system (Appendix A.2) confirms these trends.

4.3 How Good Is It to Avoid Evictions?

With a basic understanding of the factors influencing performance, we can explore some key questions. Figure 6 provides a high-level roadmap for the upcoming sections. We begin by comparing the original schedulers in Table 1 to their eviction-free versions. We use $O = B = 1024$, a scenario with frequent evictions, since in other cases, eviction-free schedulers perform similarly to their original counterparts.

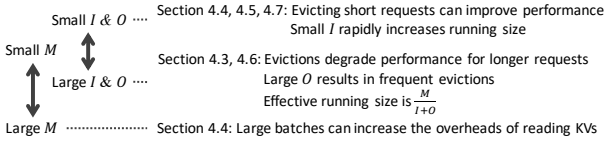


Figure 6. An overview of key insights in upcoming sections.

As illustrated in Figure 7, eviction-free schedulers generally achieve better system performance, with TPS improvements over their original versions reaching up to 6.9%, 1.7%, and 3.1% for vLLM, SARATHI, and SARATHI$_{P=C}$. However, eviction-free schedulers exhibit higher TTFT due to the need to wait for running requests to complete and release their KVs. This TTFT increase is substantial – up to 1800x for vLLM and 91.7% for SARATHI – but is offset by lower TPOT, with reductions of up to 13x for vLLM and 6.5x for SARATHI. Thus, the general TTFT-TPOT trade-off remains consistent even for eviction-free schedulers.

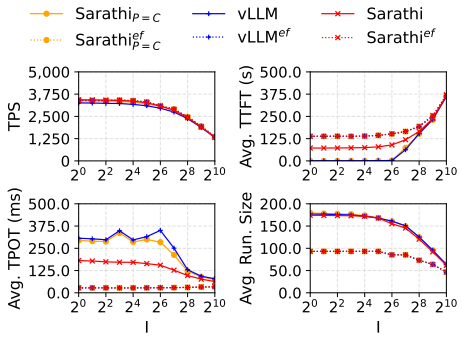


Figure 7. Results for $O = B = 1024$. Eviction-free schedulers are indicated by the *ef* superscript in the labels.

Remark. Based on the results from the previous section, it is crucial to limit the number of running requests by considering M, I, and O. The *effective* running size can be approximated as $\frac{M}{I+O}$, as supported by Figure 7, where eviction-free schedulers achieve an average running size close to $\frac{100K}{1+1024} \approx 98$ and $\frac{100K}{1024+1024} \approx 49$.

4.4 Is Increasing M a Silver Bullet?

To avoid evictions and maximize the effective running size, one might wonder if simply increasing M to a sufficiently large value could solve all issues, especially with state-of-the-art GPUs. To explore this, we vary M from 100 to 1M, testing under different memory contention levels and model-hardware configurations to simulate lower memory budgets. Figure 8 shows results for $O = 32$ and $B = 1024$.

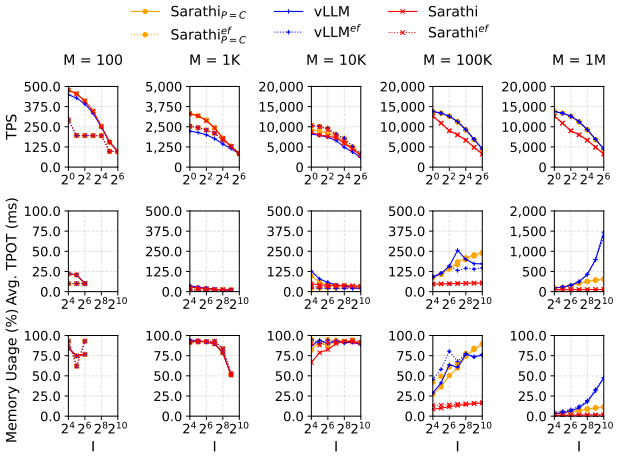


Figure 8. Results for M = 100, 1K, 10K, 100K, and 1M with $O = 32$ and $B = 1024$. The x-axes are cropped to display only the areas of interest.

Interestingly, when M is small, eviction-free schedulers actually show lower TPS compared to those allowing evictions, as they wait too long for requests to release memory.

In this context, evictions can surprisingly boost TPS and reduce latency, a phenomenon we further explore with the CSP in Section 4.5. As seen in Figure 8, evictions increase TPS by up to 2.2x and 2.3x for vLLM and SARATHI, with M = 100, and by 1.2x and 1.3x with M = 1K, mainly due to reduced TTFT, similar to the trends observed between vLLM and SARATHI in previous sections. However, these gains come with higher TPOT as a trade-off. In real inference systems (Appendix A.2), evictions yield TPS increases of 1.2x to 2.3x. However for large M values, such as 10K, evictions reduce TPS by up to 1.5x and 1.3x for vLLM and SARATHI in Figure 8.

For small M, vLLM shows lower TPS than SARATHI when I is small, as the rapid increase in running requests leads to frequent evictions. However, vLLM surpasses SARATHI once M reaches a sufficiently large size.

An intriguing effect occurs with large M values, like 100K and 1M: TPOT for vLLM increases as M and I grow. For vLLM, approximately M/I running requests accumulate to fill the KV cache. Thus, the initial decode batch has a size of M/I, requiring M KVs to be read, which imposes a heavy load on the decode steps as M grows. With larger I, the waiting time for other prefill requests to complete also increases. Before the first decode batch, the system processes about M/P prefill batches, each taking $O(P \cdot I)$ time, due to P/I requests and the quadratic complexity of prefill operations. As a result, the total prefill time is $O(M \cdot I)$. With sufficiently large M, evictions are minimized even for small I, causing each decode batch to quickly add M/I KVs to memory, which, for smaller M, would instead lead to frequent evictions and also increased TPOT.

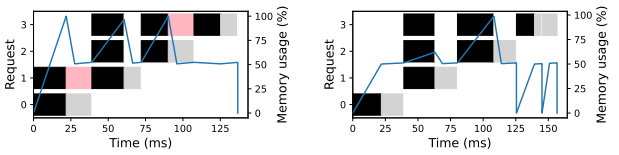
Remark. Increasing M alone is not a complete solution. Enhanced memory bandwidth is also essential to minimize the cost of reading KVs, as this factor contributes to the linear term in the decode cost model (Section 3.2). Otherwise, limiting the running size with R_{max} and, consequently, the number of KVs to read, is necessary to prevent costly decode steps. Additionally, Figure 8 shows that even with M = 1M, SARATHI and SARATHI$_{P=C}$ utilize less than 20% of M on average, even when handling numerous long requests, highlighting potential underutilization.

4.5 Are Evictions Always Evil?

As noted in the previous section, evictions can sometimes enhance system performance by reducing TTFT, albeit at the expense of increased TPOT. We use our CSP model from Section 3.3 to validate this observation.

For simplicity, we set $O = B = 4$ and vary I from 1 to 1024, with M defined as $max(2I, I + O - 1)$. This configuration allows schedulers to initiate prefills for two requests (up to $2I$) while ensuring that only one request can retain its KVs to generate the final token (limited by $I + O - 1$). The objective is to minimize latency.

Interestingly, the CSP model also opts to evict requests, similar to non-eviction-free schedulers. For $I < 128$, CSP chooses to evict short requests, as illustrated in Figure 9(a), because it is generally faster to begin generating tokens with prefills and then evict requests in later batches rather than wait for enough memory to be released. This approach can reduce latency by up to 35.2% compared to eviction-free schedules as I decreases. However, for $I \geq 128$, CSP avoids evictions, as the refill cost grows quadratically with prompt size. Here, eviction-free schedules can reduce latency by up to 27.5% compared to those with evictions as I increases. Among the schedulers, vLLM and its eviction-free version exhibit latency results close to CSP's results, with evictions for $I < 128$ and without for $I \geq 128$.



(a) $I = 64$. (b) $I = 128$.

Figure 9. CSP results for $O = B = 4$ and $M = max(2I, I + O - 1)$. Black, gray, and red boxes indicate requests in the prefill, decode, and evict phases, respectively. Blue lines represent memory (KV cache) usage over time.

Remark. Processing short requests early, even if they are subsequently evicted, can improve system performance. However, evicting long requests degrades performance due to high refill costs. Back in Figure 8, one caveat is that small M does not inherently speed up evictions but rather triggers evictions for short requests. Therefore, rather than waiting for memory to become available, scheduling short requests immediately, even without caching their KVs, may improve inference efficiency. Additionally, it is important to consider future refill costs when evicting requests to avoid evicting longer ones, suggesting that a *cost-based* approach to scheduling is essential for optimizing performance.

4.6 How Large Should R_{max} Be?

The previous results show that it is crucial to limit the number of running requests to avoid evicting long requests and processing large, KV-heavy batches. Here, M serves as a long-term cap on the total tokens in running requests, while P acts as a short-term limit on growth (or gradient). To further prevent system overload, inference systems often enforce a strict limit, R_{max}, on the number of active requests (Kwon et al., 2023; Agrawal et al., 2024a; 2023).

We vary R_{max} from 32 to 512 to determine the value that

optimizes performance. For $O = 32$ in Figure 10, TPS continues to increase with R_{max}, while for $O = 1024$, it reaches a peak at $R_{max} = 128$. Beyond this point, higher R_{max} values overload the system, causing TPS to decrease as batch sizes grow with R_{max}.

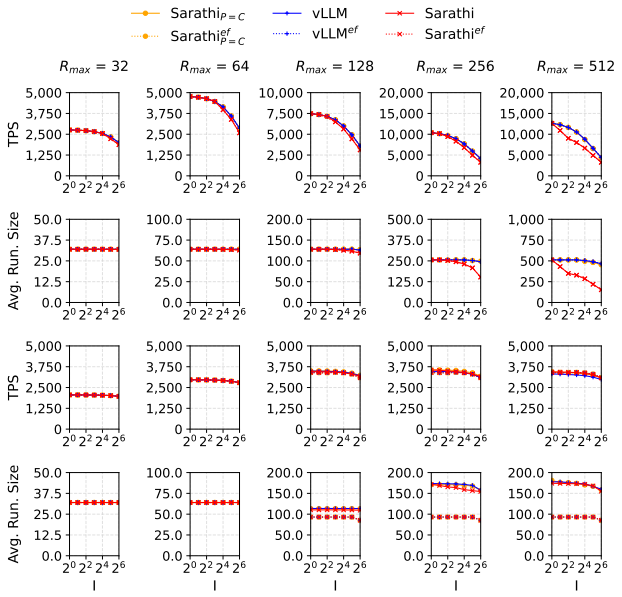


Figure 10. Results on R_{max} = 32, 64, 128, 256, and 512 with $B = 1024$. The upper two rows represent $O = 32$, while the lower two rows represent $O = 1024$.

Remark. Overall, performance improves when the running size aligns with the effective running size observed in eviction-free schedulers, which approximates $\frac{M}{I+O}$ in Figure 10. Therefore, R_{max} should be high for small I and O, but lower for larger values of I and O. Both overloading and underloading requests result in suboptimal performance.

4.7 Should We Prioritize Requests with Small I or O?

Until now, we have assumed uniform input and output sizes for all requests. In the case of heterogeneous requests, prioritizing those with fewer steps, i.e., smaller output sizes, is known to improve scheduling efficiency (Fu et al., 2024; Qiu et al., 2024; Zheng et al., 2023b). However, we question this approach, finding that input size can be an equally or more important factor, as discussed in Appendix A.3 due to space limitations.

5 Conclusion and Vision

In this work, we propose an analytical framework to compare schedulers and benchmark them against optimal scheduling policies by leveraging cost models, providing a more scalable alternative to exhaustive GPU-based testing for every scheduler and workload configuration. By incorporating constraint satisfaction problem (CSP) techniques, our approach theoretically identifies scheduling inefficiencies, showing that 30% of latency can still be optimized. Our findings highlight that an analytical focus on the effects of evictions and memory demands can lead to significant performance gains.

Building on our analysis and CSP formulation, we envision a unified cost model adaptable across devices, storage hierarchies, model types, and inference systems. This model would support optimal policy discovery via CSP across diverse workloads and adaptive cost-based scheduling, leading to more efficient development cycles and scalable inference systems, with potential reductions in CO2 emissions and operating costs.

Hardware and model characteristics determine constants like M and C as well as cost model coefficients and biases (Section 3.2), opening opportunities for policies like prefill-decode disaggregation (Strati et al., 2024; Zhong et al., 2024b; Patel et al., 2024b) and offloading KVs to CPUs or flash storage (Lee et al., 2024; Pan et al., 2024). Future work could explore new hardware capabilities, parallelization strategies (Lie, 2024), model quantization (Dettmers et al., 2022; Ma et al., 2024), KV compression (Liu et al., 2024), and attention models optimized for reduced data loads (Hu et al., 2022; Gu et al., 2022; Beltagy et al., 2020).

Expanding to more complex scenarios beyond offline and online workloads will also be crucial as inference patterns grow in complexity, including prefix-sharing, multi-path reasoning, and compound AI systems (Patel et al., 2024a; Lin et al., 2024; Zheng et al., 2023a; Jeong et al., 2024; Zhong et al., 2024a; Zaharia et al., 2024).

To develop more effective scheduling policies and cost models, RL could be used to generate and refine candidate policies/models against estimated or real costs, while comparing with scaled CSP solutions to bridge any gaps in performance, reducing the need for manual optimization.

Overall, our work and vision aim to empower LLM service providers and developers to build cost-effective and environmentally sustainable inference systems without extensive resource expenditure.

References

Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. SARATHI: efficient LLM inference by piggybacking decodes with chunked prefills. *CoRR*, abs/2308.16369, 2023. doi: 10.48550/ARXIV.2308.16369. URL https://doi.org/10.48550/arXiv.2308.16369.

Agrawal, A., Kedia, N., Mohan, J., Panwar, A., Kwatra, N., Gulavani, B. S., Ramjee, R., and Tumanov, A. VIDUR:

A large-scale simulation framework for LLM inference. In Gibbons, P. B., Pekhimenko, G., and Sa, C. D. (eds.), *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*. mlsys.org, 2024a.

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., Tumanov, A., and Ramjee, R. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In Gavrilovska, A. and Terry, D. B. (eds.), *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pp. 117–134. USENIX Association, 2024b.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020. URL https://arxiv.org/abs/2004.05150.

Bertsimas, D. and Tsitsiklis, J. N. *Introduction to linear optimization*, volume 6 of *Athena scientific optimization and computation series*. Athena Scientific, 1997. ISBN 978-1-886529-19-9.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J. G., Le, Q. V., and Salakhutdinov, R. Transformer-xl: Attentive language models beyond a fixed-length context. In Korhonen, A., Traum, D. R., and Màrquez, L. (eds.), *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pp. 2978–2988. Association for Computational Linguistics, 2019. doi: 10.18653/V1/P19-1285. URL https://doi.org/10.18653/v1/p19-1285.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Gpt3.int8(): 8-bit matrix multiplication for transformers at scale. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

Fu, Y., Zhu, S., Su, R., Qiao, A., Stoica, I., and Zhang, H. Efficient LLM scheduling by learning to rank. *CoRR*, abs/2408.15792, 2024. doi: 10.48550/ARXIV.2408.15792. URL https://doi.org/10.48550/arXiv.2408.15792.

Gu, A., Goel, K., and Ré, C. Efficiently modeling long sequences with structured state spaces. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Jeong, S., Baek, J., Cho, S., Hwang, S. J., and Park, J. Adaptive-rag: Learning to adapt retrieval-augmented large language models through question complexity. In Duh, K., Gómez-Adorno, H., and Bethard, S. (eds.), *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, pp. 7036–7050. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.NAACL-LONG.389. URL https://doi.org/10.18653/v1/2024.naacl-long.389.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In Flinn, J., Seltzer, M. I., Druschel, P., Kaufmann, A., and Mace, J. (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Lee, W., Lee, J., Seo, J., and Sim, J. Infinigen: Efficient generative inference of large language models with dynamic KV cache management. In Gavrilovska, A. and Terry, D. B. (eds.), *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pp. 155–172. USENIX Association, 2024.

Lie, S. Inside the cerebras wafer-scale cluster. *IEEE Micro*, 44(3):49–57, 2024. doi: 10.1109/MM.2024.3386628. URL https://doi.org/10.1109/MM.2024.3386628.

Lin, C., Han, Z., Zhang, C., Yang, Y., Yang, F., Chen, C., and Qiu, L. Parrot: Efficient serving of llm-based applications with semantic variable. In Gavrilovska, A. and

Terry, D. B. (eds.), *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pp. 929–945. USENIX Association, 2024.

Liu, Y., Li, H., Cheng, Y., Ray, S., Huang, Y., Zhang, Q., Du, K., Yao, J., Lu, S., Ananthanarayanan, G., Maire, M., Hoffmann, H., Holtzman, A., and Jiang, J. Cachegen: KV cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM 2024, Sydney, NSW, Australia, August 4-8, 2024*, pp. 38–56. ACM, 2024. doi: 10.1145/3651890.3672274. URL https://doi.org/10.1145/3651890.3672274.

Luccioni, A. S., Viguier, S., and Ligozat, A.-L. Estimating the carbon footprint of bloom, a 176b parameter language model. *Journal of Machine Learning Research*, 24 (253):1–15, 2023. URL https://www.jmlr.org/papers/volume24/23-0069/23-0069.pdf.

Ma, S., Wang, H., Ma, L., Wang, L., Wang, W., Huang, S., Dong, L., Wang, R., Xue, J., and Wei, F. The era of 1-bit llms: All large language models are in 1.58 bits. *CoRR*, abs/2402.17764, 2024. doi: 10.48550/ARXIV.2402.17764. URL https://doi.org/10.48550/arXiv.2402.17764.

Pan, X., Li, E., Li, Q., Liang, S., Shan, Y., Zhou, K., Luo, Y., Wang, X., and Zhang, J. Instinfer: In-storage attention offloading for cost-effective long-context LLM inference. *CoRR*, abs/2409.04992, 2024. doi: 10.48550/ARXIV.2409.04992. URL https://doi.org/10.48550/arXiv.2409.04992.

Patel, L., Jha, S., Guestrin, C., and Zaharia, M. LOTUS: enabling semantic queries with llms over tables of unstructured and structured data. *CoRR*, abs/2407.11418, 2024a. doi: 10.48550/ARXIV.2407.11418. URL https://doi.org/10.48550/arXiv.2407.11418.

Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., and Bianchini, R. Splitwise: Efficient generative LLM inference using phase splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*, pp. 118–132. IEEE, 2024b. doi: 10.1109/ISCA59077.2024.00019. URL https://doi.org/10.1109/ISCA59077.2024.00019.

Pistikopoulos, E. N. C.A. floudas, nonlinear and mixed-integer optimization. fundamentals and applications. *J. Glob. Optim.*, 12(1):108–110, 1998. doi: 10.1023/A:1008256302713. URL https://doi.org/10.1023/A:1008256302713.

Qiu, H., Mao, W., Patke, A., Cui, S., Jha, S., Wang, C., Franke, H., Kalbarczyk, Z. T., Basar, T., and Iyer, R. K. Efficient interactive LLM serving with proxy model-based sequence length prediction. *CoRR*, abs/2404.08509, 2024. doi: 10.48550/ARXIV.2404.08509. URL https://doi.org/10.48550/arXiv.2404.08509.

Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020. ISBN 9780134610993. URL http://aima.cs.berkeley.edu/.

Schrijver, A. *Theory of Linear and Integer Programming*. Wiley-Interscience, 1998.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. Access path selection in a relational database management system. In Bernstein, P. A. (ed.), *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, pp. 23–34. ACM, 1979. doi: 10.1145/582095.582099. URL https://doi.org/10.1145/582095.582099.

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In Bartlett, P. L., Pereira, F. C. N., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pp. 2960–2968, 2012.

Strati, F., McAllister, S., Phanishayee, A., Tarnawski, J., and Klimovic, A. Déjàvu: Kv-cache streaming for fast, fault-tolerant generative LLM serving. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=AbGbGZFYOD.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017.

Xu, J., Zhang, R., Guo, C., Hu, W., Liu, Z., Wu, F., Feng, Y., Sun, S., Shao, C., Guo, Y., Zhao, J., Zhang, K., Guo, M., and Leng, J. vtensor: Flexible virtual tensor management for efficient LLM serving. *CoRR*, abs/2407.15309, 2024. doi: 10.48550/ARXIV.2407.15309. URL https://doi.org/10.48550/arXiv.2407.15309.

Yu, G., Jeong, J. S., Kim, G., Kim, S., and Chun, B. Orca: A distributed serving system for transformer-based generative models. In Aguilera, M. K. and Weatherspoon, H. (eds.), *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pp. 521–538. USENIX Association, 2022.

Yuan, Z., Shang, Y., Zhou, Y., Dong, Z., Zhou, Z., Xue, C., Wu, B., Li, Z., Gu, Q., Lee, Y. J., Yan, Y., Chen, B., Sun, G., and Keutzer, K. LLM inference unveiled: Survey and roofline model insights. *CoRR*, abs/2402.16363, 2024. doi: 10.48550/ARXIV.2402.16363. URL https://doi.org/10.48550/arXiv.2402.16363.

Zaharia, M., Khattab, O., Chen, L., Davis, J. Q., Miller, H., Potts, C., Zou, J., Carbin, M., Frankle, J., Rao, N., and Ghodsi, A. The shift from models to compound ai systems. *The Berkeley Artificial Intelligence Research Blog*, Feb 2024. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/.

Zheng, L., Yin, L., Xie, Z., Huang, J., Sun, C., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C. W., and Sheng, Y. Efficiently programming large language models using sglang. *CoRR*, abs/2312.07104, 2023a. doi: 10.48550/ARXIV.2312.07104. URL https://doi.org/10.48550/arXiv.2312.07104.

Zheng, Z., Ren, X., Xue, F., Luo, Y., Jiang, X., and You, Y. Response length perception and sequence scheduling: An llm-empowered LLM inference pipeline. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023b.

Zhong, T., Liu, Z., Pan, Y., Zhang, Y., Zhou, Y., Liang, S., Wu, Z., Lyu, Y., Shu, P., Yu, X., Cao, C., Jiang, H., Chen, H., Li, Y., Chen, J., Hu, H., Liu, Y., Zhao, H., Xu, S., Dai, H., Zhao, L., Zhang, R., Zhao, W., Yang, Z., Chen, J., Wang, P., Ruan, W., Wang, H., Zhao, H., Zhang, J., Ren, Y., Qin, S., Chen, T., Li, J., Zidan, A. H., Jahin, A., Chen, M., Xia, S., Holmes, J., Zhuang, Y., Wang, J., Xu, B., Xia, W., Yu, J., Tang, K., Yang, Y., Sun, B., Yang, T., Lu, G., Wang, X., Chai, L., Li, H., Lu, J., Sun, L., Zhang, X., Ge, B., Hu, X., Zhang, L., Zhou, H., Zhang, L., Zhang, S., Liu, N., Jiang, B., Kong, L., Xiang, Z., Ren, Y., Liu, J., Jiang, X., Bao, Y., Zhang, W., Li, X., Li, G., Liu, W., Shen, D., Sikora, A., Zhai, X., Zhu, D., and Liu, T. Evaluation of openai o1: Opportunities and challenges of AGI. *CoRR*, abs/2409.18486, 2024a. doi: 10.48550/ARXIV.2409.18486. URL https://doi.org/10.48550/arXiv.2409.18486.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In Gavrilovska, A. and Terry, D. B. (eds.), *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pp. 193–210. USENIX Association, 2024b.

Zhu, K., Zhao, Y., Zhao, L., Zuo, G., Gu, Y., Xie, D., Gao, Y., Xu, Q., Tang, T., Ye, Z., et al. Nanoflow: Towards optimal large language model serving throughput. *arXiv preprint arXiv:2408.12757*, 2024.

A APPENDIX

A.1 Analysis Under Low Contention

With low contention ($B = 32$), no evictions occur across all schedulers. We categorize the schedulers from Table 1 into $\{\text{SARATHI}_{nohy}\}$, $\{\text{SARATHI}\}$, and others based on their performance. Figure 11 presents results for SARATHI_{nohy}, SARATHI, and VLLM as representative examples.

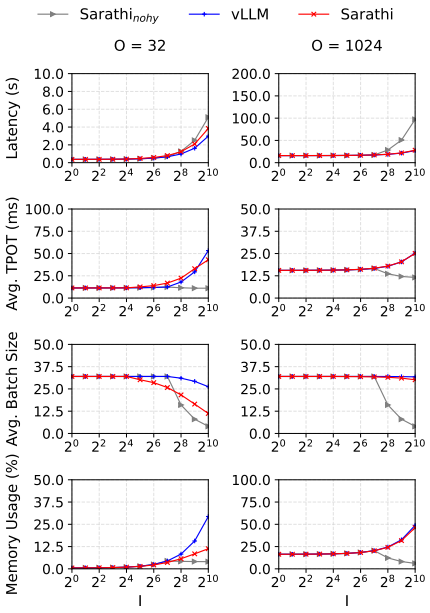


Figure 11. Results for small $B = 32$. Each column represents a specific O, with I shown on the x-axis.

As discussed in Section 4.2, latency increases across all schedulers as I and O grow. VLLM consistently achieves the lowest latency by prioritizing prefills and batching as many decodes as possible, resulting in the largest batch sizes, efficient processing, and reduced latency without evictions. Other schedulers in the same group as VLLM also have $P = C = 4096$, yielding similar prefill speeds and comparable performance.

For $O = 32$, SARATHI experiences higher latency and TPOT than VLLM due to smaller batch sizes, as it processes only $P = 512$ prefill tokens at a time, which reduces the number of decode requests in each batch. However, for $O = 32$ and $B = 1024$, SARATHI achieves a lower TPOT than VLLM, as VLLM reads more KVs from the cache, resulting in heavier decode batches. For $O = 1024$, SARATHI's prefill speed catches up with VLLM, resulting in similar batch sizes and comparable performance.

SARATHI_{nohy} exhibits a unique pattern, with high latency as I increases. This is due to a significant decrease in batch size for the following reasons: (1) each prefill batch can add up to C/I requests, which decreases as I grows, (2) if any running request is in the decode phase, it triggers a decode-only batch, whose size is limited by (1), and (3) these decode-only batches persist until all running requests complete.

Remark. With low contention, evictions do not occur, and latency primarily depends on batch size, prefill speed, and the efficiency of batched decodes. VLLM achieves the fastest prefill speed by prioritizing prefills and processing up to C prefill tokens per batch. Larger batches can reduce TPOT, although an increase in the number of KVs may increase TPOT.

A.2 Results on a Real Inference System

This section presents results from running the experiments in Sections 4.2 and 4.4 on a real inference system (Figures 12 and 13). We used VLLM (Kwon et al., 2023) v0.6.3 with the same setup as in Section 4: one A100 GPU and the Llama-2-7B model. For clarity, VLLM refers to the scheduler as before, while VLLM-SYS denotes the inference system.

Figure 12 shows that the three schedulers perform similarly to their behavior in Figure 4 on INFERMAX, although the relative error reaches 30%, higher than the 9% reported in (Agrawal et al., 2024a). This error decreases to 3% as O increases. Possible reasons include: (1) with smaller O, the request footprint of batches changes rapidly, leading to greater variance in batch times, and (2) cumulative errors in the cost model across batches. Besides refining cost models as discussed in Section 3.2, a detailed analysis of discrepancies between INFERMAX, VIDUR, and VLLM-SYS, and reducing these gaps, is left for future work.

Recent inference systems, e.g., (Zhu et al., 2024), also exploit overlapping scheduling and GPU operations to reduce non-GPU processing time, which could further improve results in Figure 12.

Figure 13 shows results for varying M. Consistent with Section 4.4, evictions increase TPS for smaller M values of 100 and 1K, with TPS increasing up to 2.1x and 2.3x for VLLM and SARATHI under $M = 100$, and up to 1.2x and 1.4x under M = 1K. For M = 10K, evictions decrease TPS by up to 1.5x and 1.08x for VLLM and SARATHI.

These results demonstrate that our approach of analyzing existing schedulers and leveraging CSP to hypothesize optimal policies reveals effective scheduling strategies.

A.3 How Should We Prioritize Heterogeneous Requests?

This section presents the performance results of ranking-based schedulers that prioritize requests by short input or output sizes rather than by phase (prefill or decode), especially for heterogeneous requests with varying input and

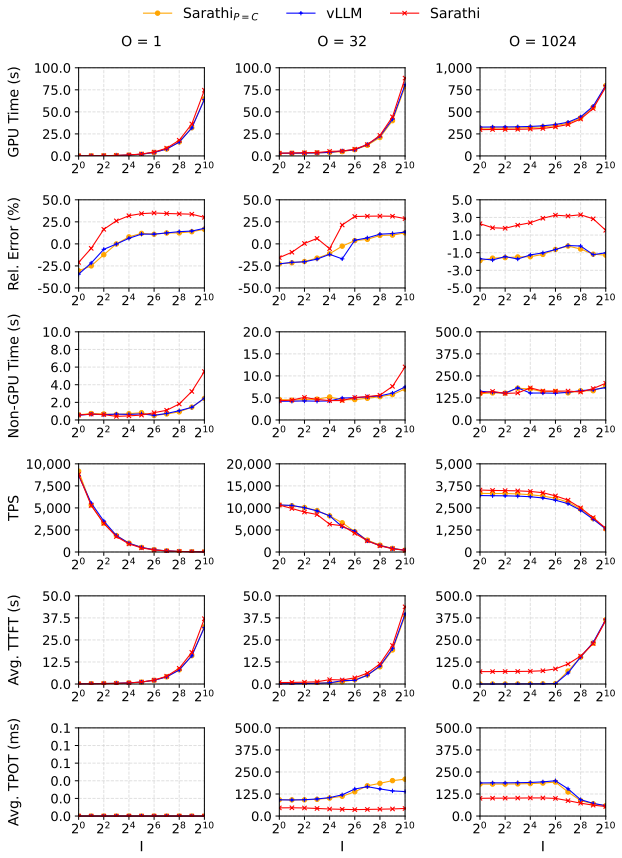


Figure 12. Results of running the schedulers in Figure 4 on vLLM-SYS with $B = 1024$. GPU time represents the duration for GPU-based operations, including matrix multiplications and attentions, while non-GPU time covers scheduling and token sampling. The second row indicates the relative error of the latency in Figure 4 compared to the actual GPU time recorded here. TPS, TTFT, and TPOT are calculated using GPU times only.

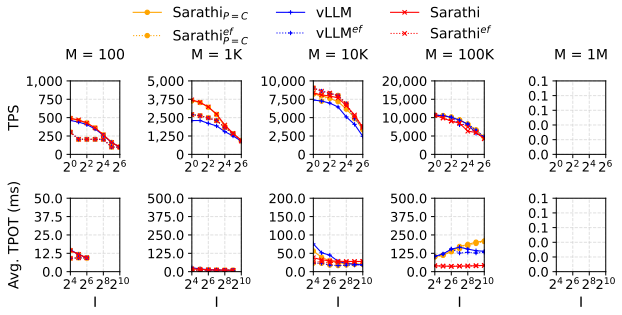


Figure 13. Results of running the schedulers in Figure 8 on vLLM-SYS with $O = 32$ and $B = 1024$. No results are provided for M = 1M since the maximum M available on the physical A100 GPU with the Llama-2-7B model is below 1M.

output sizes. Using our Algorithm 1 from Section 3.1, we implement ranking by grouping requests in GROUPREQUESTS (1) and ordering by input or output size, using a single group here.

Regarding the workloads, we generate each workload as a mix of two types of request groups (with $S = \{8, 16\}, L = \{512, 1024\}$):

- Short I and short O (SISO): $I \in S, O \in S$
- Short I and long O (SISO): $I \in S, O \in L$
- Long I and short O (SISO): $I \in L, O \in S$
- Long I and long O (SISO): $I \in L, O \in L$

The requests in each workload are randomly shuffled, and the number of requests B varies from 16 to 1024. We compare four schedulers:

- $Rank_{org}$: No prioritization by I or O.
- $Rank_I$: Requests are prioritized by I.
- $Rank_O$: Requests are prioritized by O.

Latency. Figure 14 shows latency for six workloads with different query mixes. Prioritizing requests with smaller I can reduce latency, as seen in Section 4.5. When the KV cache reaches M and frequent evictions occur as B grows, scheduling smaller I requests first, despite evictions and refills, can reduce latency. Consequently, $Rank_I$ shows up to 1.3x and 1.4x lower latency than $Rank_O$ and $Rank_{org}$, particularly in the LILO+SILO workload (top middle in Figure 14) with frequent evictions.

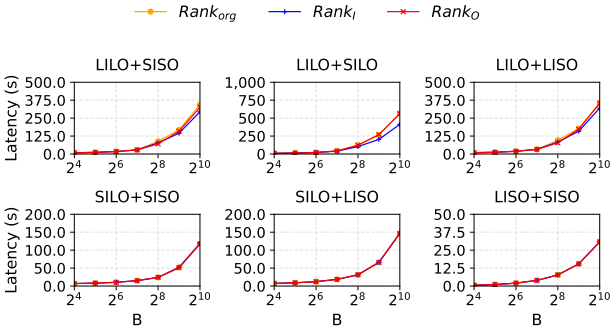


Figure 14. Latency over heterogeneous workloads.

TTFT. Scheduling small I or O requests first can reduce TTFT by prioritizing short-running requests. For example, assume two requests $R1$ and $R2$ with different I values, where their (prefill time, decode time) pairs are (1, 1) and (2, 1). Scheduling $R1$ first yields an average TTFT of $(1 + (1 + 1 + 2))/2 = 5/2$, while scheduling $R2$ first results in $(2 + (2 + 1 + 1))/2 = 6/2$. If the (prefill time, decode time) pairs are $(1, 1)$ and $(1, 2)$, then scheduling $R1$ first results in the average TTFT of $(1 + (1 + 1 + 1))/2 = 4/2$, where scheduling $R2$ first results in $(1 + (1 + 2 + 1))/2 = 5/2$.

As Figure 15 shows, $Rank_I$ achieves up to 3x and 2x lower TTFT than $Rank_O$ and $Rank_{org}$ for all workloads except LILO+LISO, where requests have similar I. In this case, $Rank_O$ achieves the lowest TTFT, up to 1.6x and 2.1x lower than $Rank_I$ and $Rank_{org}$.

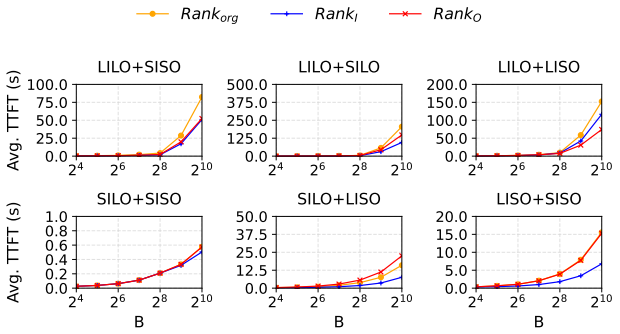


Figure 15. Average TTFT over heterogeneous workloads.

TPOT. When small I requests are prioritized, TPOT decreases as it reduces the time of requests in the eviction phase. Prioritizing small I or O requests can lower TPOT by minimizing evictions or avoiding heavy batches. If we prioritize requests with small I, TPOT decreases since it reduces the time of requests in the eviction phase. Assume that we process long requests with large I and short requests with small I together where long requests have higher priority, in contrast to $Rank_I$. If the memory is limited (e.g., in workloads containing LILO), evictions occur for the short requests, and they need to wait for long requests to release their memory. This increases TPOT. In contrast, prioritizing short requests leads to a lower waiting time for long requests. This is why in Figure 16 $Rank_I$ has up to 1.3x smaller TPOT than $Rank_O$ and $Rank_{org}$, on the workloads that contain LILO and evictions occur frequently under insufficient memory.

However, prioritizing small I may also increase TPOT as the batch size increases (the same reason for SARATHI$_{P=C}$ having larger batches and TPOT than SARATHI in Section 4, that up to P/I requests can be added to the running requests per batch). $Rank_O$ shows up to 1.7x and 1.3x smaller TPOT than $Rank_I$ and $Rank_{org}$ on the workloads that do not contain LILO.

Remark. Prioritizing requests with smaller I generally yields better performance compared to prioritizing by O, except when (1) requests have similar I but highly variable O, or (2) memory is sufficient (minimizing evictions), and TPOT is critical, as in workloads without LILO. In such cases, ordering by O is more effective. This is notable since most request-ranking studies have focused on output size (Qiu et al., 2024; Zheng et al., 2023b; Fu et al., 2024), we suggest that input size may be an equally or more significant factor.

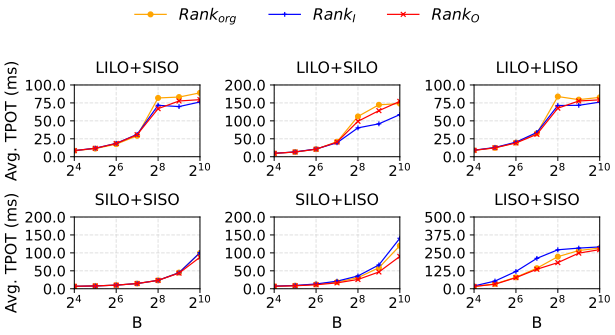


Figure 16. Average TPOT over heterogeneous workloads.