Prophet: An LLM Inference Engine Optimized For Head-of-Line Blocking

Schwinn Saereesitthipitak, Ashish Rao, Cathy Zhou, William Li

Abstract

We design, implement, and benchmark a novel strategy for scheduling inference requests for large language models (LLMs). Traditional LLM inference engines schedule at a per-request level in a first-come-first-serve manner, which leads to head-of-line blocking, degrading latency for short requests. Our proposed strategy 1) employs iteration-level scheduling, 2) disaggregates the "prefill" and "decode" stages of requests across different machines, and 3) uses priority-based schedulers. Prefill-stage requests are scheduled by the shortest remaining processing time (SRPT) method, and decode-stage requests are scheduled using the MLFQ algorithm. We experiment using the estimated number of decode iterations remaining through zero-shot prompting as "skip-join" MLFQ priorities. We evaluate the system on three key latency metrics: time to first token (TTFT), time per output token (TPOT), and job completion time (JCT). We demonstrate how iteration level scheduling, disaggregated request processing, and decode length prediction can be combined to serve LLM inference requests with low latency.

1 Introduction and Prior Work

The use of LLMs in applications like chatbots, search engines, and coding assistants has driven widespread interest in efficiently serving inference requests. To serve a large userbase, LLM inference engines must be optimized for high throughput and latency. At a high-level, the language modeling task consists of recursively predicting the most likely next token from the input. LLMs by and large use the Attention mechanism [10] to compute the most likely next token. If naively implemented, language models run in quadratic time in the number of input tokens per prediction step, which hinders its scalability.

To mitigate this, most LLM inference pipelines use KV cache optimization [8] to bring down the computational complexity of language modeling. In particular, since the K and V matrices of previous prediction steps never change, they are stored rather than recomputed. With KV caching, the language modeling task can be decomposed into two steps: prefill and decode. In the prefill step, the KV cache for the prompt is generated using Self-Attention. In the decode step, new tokens are predicted recursively using Attention on the last token's encoding and all previous KV cache entries.

Traditional LLM inference engines like Triton [7], Megatron-LM [9] and FasterTransformer [**?**] wait until a batch of requests of a specific size is available, and then run both prefill and decode stages to completion on that batch before returning the model outputs. Different requests may have vastly different lengths, for instance due to the model generating an 'end-of-sequence' token. In this case, short requests idly occupy the slot in the batch until the longest requests completes, deteriorating both throughput and latency.

Orca [12] proposes iteration-level scheduling, where batches of requests are constructed before the prefill stage and at each decode iteration. This enables completed requests to be evicted from the batch as soon as they are complete, and for new requests to be inserted as soon as they are available. Orca uses a simplistic first-come-first-serve (FCFS) scheduler to schedule which requests should be inserted into a batch. However, FCFS scheduling is known to cause head-of-line blocking as short requests must wait for longer requests to complete,

thereby inflating inference latency on short requests.

FastServe [11] addresses head-of-line blocking by using a Skip-Join Multi-Level Feedback Queue (MLFQ) scheduler. Requests are partitioned into different queues of different priorities, and high priority requests are scheduled first, though starved requests are periodically promoted. Therefore, long-running requests are preempted, allowing shorter requests to run.

FastServe and similar works like SARATHI [1] still suffer from head-of-line blocking due to interference between prefill and decode iterations, since prefills and decodes cannot be batched together, and have vastly different execution time. To mitigate this, DistServe [14] proposes disaggregation of prefill and decode stages on different GPUs.

Furthermore, the prefill stage tends to be compute bound, so a smaller batch size and intra-op parallelism works best. Conversly, the decode stage tends to be memory bound, so a larger batch size and inter-op/pipeline parallelism works best. Disaggregation allows for such stage-specific optimizations. Nonetheless, DistServe [14] still uses an FCFS scheduling strategy like Orca [12], leading to head-of-line blocking within each request stage.

2 Methodology

Inspired by the previous body of work in LLM inference engines, we build *Prophet*, an LLM inference engine optimized for mitigating head-of-line blocking. Prophet supports iteration-level scheduling and disaggregation of prefills and decodes. We adopt an SRPT scheduler for the prefill stage and a MLFQ scheduler for the decode stage (both with starvation prevention) to reduce head-of-line blocking. We implement the LLM inference engine with Ray [6].

We evaluate the performance of our schedulers by measuring time to first token (TTFT), and time per output token (TPOT) and job completion time (JCT) across varying prompt and output lengths.

3 Architecture

Prophet (Figure 1) consists of the following Ray actors:

- Request Generator: takes requests from a client and passes requests to the Prefiller
- Prefiller: runs a prefill process on a GPU.
- Decoder: runs the decoder process on a GPU.
- Output Consumer: takes outputs from the Decoder and sends them to the client
- KV Coordinator: coordinates transfer of KV cache from prefill to decode actors

When a request first arrives at a Prefiller, it is added to the Prefiller's local scheduler. Each iteration, the Prefiller gathers requests one batch at a time from its scheduler, prefills the KV caches for all requests in the batch, and stores the results in a KV Cache Manager. The KV Cache Manager lives on the Prefiller and occupies the same GPU memory, holding the KV caches until the Decoder requests them. The request is put in a "pending" queue that the Decoder can pull from.

Once the Decoder pops a request from the queue, it asks the KV Coordinator to coordinate a transfer of the request's KV cache from the corresponding Prefiller. Then, the Decoder adds the request to its local scheduler. At each iteration, the Decoder schedules a batch requests to make progress on, and is able to advance each request in the batch by one token. After each iteration, all requests that finished (via EOS token or maximum generation length) have their KV cache cleared and the output is sent to the Output Consumer.

Note that the KV cache of the currently executing decode batch is only allocated once at initialization to ensure it is contiguous in memory. When requests are newly scheduled or restored, their KV cache is copied in, and when they are preempted their KV cache is copied out.

There can be many Prefillers and Decoders running at any time. At initialization, every Decoder knows of the existence of all possible Prefillers. When the Decoder pulls new requests, it can choose a random Prefiller to pull from. Since each Prefiller's queue is a single-threaded Ray Actor, multiple Decoders can pull from the same Prefiller's queue at once without synchronization issues.

3.1 Prefiller Scheduler

We chose Shortest-Remaining-Processing-Time (SRPT) as a basis for the Prefiller scheduler design. In SRPT scheduling, the request that is estimated to terminate first is run first.

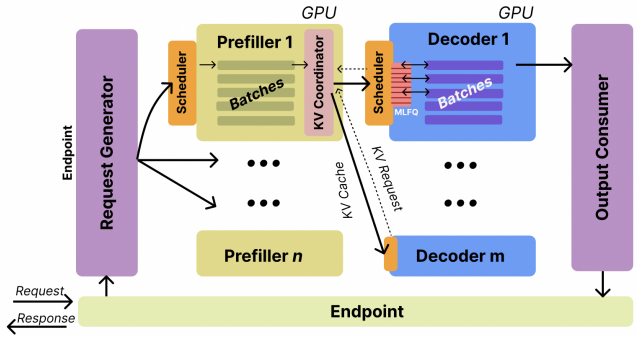

Figure 1: Overall workflow of model serving

For optimizing latency of short requests, SRPT is the most optimal choice among common scheduler paradigms but requires an accurate estimate of processing time. Previous works [14] noted that prefill time is proportional to the input length, therefore the input length can be used directly as the processing time estimate.

However, when jobs are submitted continuously, we found SRPT to severly starve long requests under high request load. Therefore, the scheduler was modified to have a starvation preventoin mechanism, where the processing time estimate would be set to 0 if the request was starved for sufficiently long. This guarantees that the request will be prefilled even under high load.

3.2 Decoder Scheduler

Since it is difficult to predict the number of output tokens beforehand, the scheduler chosen for the Decoder actor is a Multi-Level Feedback Queue (MLFQ) [4] with starvation prevention. In our scheduler, there are multiple queues of requests, each queue corresponding to a different priority. Requests with highest priority are scheduled first. When a request arrives, it is placed into the queue with the highest priority. Each queue has a time quantum, which is the maximum number of iterations a request can run in the queue before it is moved to a lower priority queue. If a request has not run for a consecutive number of iterations exceeding a starvation limit parameter, the request is promoted to the highest priority.

Note that requests do not "skip-join" the queue (as in FastServe [11]), since using the prefill length as an indication of priority does not work for a disaggregated Decoder.

3.3 Backpressure

When the KV cache of a request is transferred to a Decoder, it is materialized in the Decoder's GPU even if it is not scheduled. In fact, the memory usage of the Decoder consists of all requests "in the scheduler" (i.e. requests that the scheduler is aware of and could schedule). Some of these requests will be in the current running batch, and some will not. Therefore, the Decoder's actor thread is blocked until there are less than *max_requests_in_scheduler* requests in the scheduler. This in turn creates backpressure on the KV Cache Managers of each Prefiller, so each Prefiller's thread is also blocked until the number of requests awaiting transfer does not exceed the prefill batch size.

4 Implementation

Prophet is implemented in Python using Ray and PyTorch. For our proof-of-concept, we serve instances of Meta's instruction-tuned Llama3 8B model [2], where each GPU runs one instance of the model (i.e. no model parallelism). Since prefill throughput is inherently much higher than decode throughput, our codebase supports one Prefiller instance, but can easily be extended to more.

In the original Llama3 code, the generation strategy is to run next-token prediction for the request with the smallest start index in the batch. This means other requests can only make progress if their start index is the same, degrading throughput. We modified the generation strategy to predict the next-token for all requests in the batch, which simplifies our model for scheduling. To allow this, KV caches in decode workers are padded to the maximum sequence length that can be processed by the model.

KV caches are transferred from Prefillers to Decoders using NCCL via the Ray Collective Communication library.

5 Evaluation and Discussion

We benchmarked Prophet on one AWS p4d.12xlarge instance with 8 A100 SXM 40GB NVIDIA GPUs. We use the ShareGPT dataset [3] to benchmark the performance of Prophet because it high variance in both prompt and output lengths. We refer to [14] for the specific prompt and output length distributions. Requests are sent to the

inference engine using a Poisson arrival process with a fixed λ parameter, denoting the number of requests per second. We set the model's context size to 1024 tokens. We truncate the ShareGPT dialogs such that the encoded prompt, including the last dialog entry, is capped at 950 tokens. Each trial is run until 1000 requests are completed.

We set the Prefiller's batch size to 4 and maximum scheduler size to 16. The Prefiller's scheduler starvation limit is set to 16 iterations. Decoders have a batch size of 16 and maximum scheduler size 32. For the MLFQ scheduler, there are 5 queues with quantums 25, 50, 100, 200 and 400 respectively. The starvation limit is set to 200 iterations.

We evaluate Prophet on the mean, 50%, 90% and 99% performance using TTFT, TPOT, and JCT metrics. We define our metrics precisely as follows:

- TTFT: Time elapsed from creation of request in Request Generator until the first token arrives at the Decoder
- TPOT: Time elapsed from when the first token arrives at the Decoder until the last token arrives at the Output Consumer, divided by the number of output tokens
- JCT: Time elapsed from creation of request in Request Generator until result is received in Output Consumer

5.1 Prefill Scheduler Evaluation

The normalized TTFT (i.e. TTFT divided by the prompt length) is the most apprioriate metric to evaluate our Prefiller scheduling algorithm. In the ideal case, the normalized TTFT is uniform across prompt lengths.

Scheduler	Median	90th %ile	99th %ile
fcfs_mlfq	31.902	74.740	80.624
srpt_mlfq	22.929	70.495	100.170

Table 1: TTFT metrics at 2 requests/second for FCFS prefill and MLFQ decode scheduling (fcfs_mlfq), and SRPT prefill and MLFQ decode scheduling (srpt_mlfq).

From Fig 2 and Table 1, it is apparent that SRPT prefill scheduling significantly lowers TTFT latency metrics, particularly for the median latency. However, there is still significant head-of-line blocking at the 99th percentile as evidenced by the large normalized TTFT for

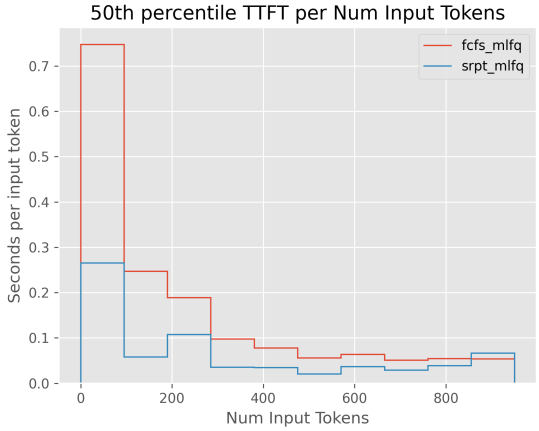

Figure 2: Median normalized TTFT at 2 requests/second for FCFS prefill and MLFQ decode scheduling (fcfs_mlfq), and SRPT prefill and MLFQ decode scheduling (srpt_mlfq).

short requests even with srpt_mlfq scheduling in Fig 3). We anticipate this may be due to the starvation prevention mechanism, which always will promote starved long prefill requests to the highest priority, blocking shorter requests in the process. Higher starvation thresholds might show srpt prefill scheduling lowering 99th percentile TTFT by larger margins.

5.2 Decode Scheduler Evaluation

Scheduler	Median	90th %ile	99th %ile
fcfs_fcfs	0.335	0.674	2.419
fcfs_mlfq	0.371	0.462	0.52
srpt_fcfs	0.3325	0.6675	2.216
srpt_mlfq	0.377	0.4625	0.550

Table 2: TPOT metrics at 2 requests/second for various scheduler configurations.

To evaluate our Decoder scheduling algorithm, we use the TPOT metric. In our implemention, TPOT includes the time waiting in the scheduler, which captures head-of-line blocking. From Table 2, Fig 4, and Fig 5, it is clear that using MLFQ decode scheduling significantly lowers tail latencies (90th and 99th percentiles). MLFQ decoding slightly increases median latency due to the overheads incurred by preempting long running requests in favor of shorter ones.

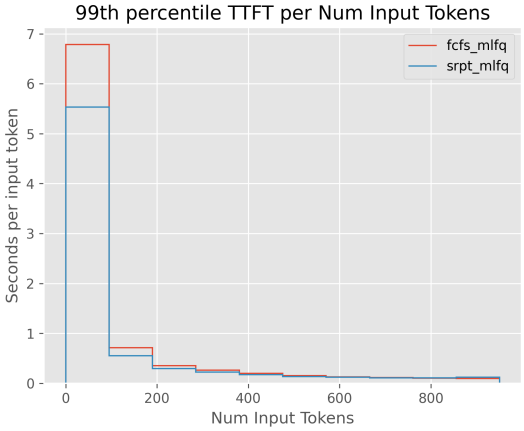

Figure 3: 99th Percentile normalized TTFT at 2 requests/second for FCFS prefill and MLFQ decode scheduling (fcfs_mlfq), and SRPT prefill and MLFQ decode scheduling (srpt_mlfq).

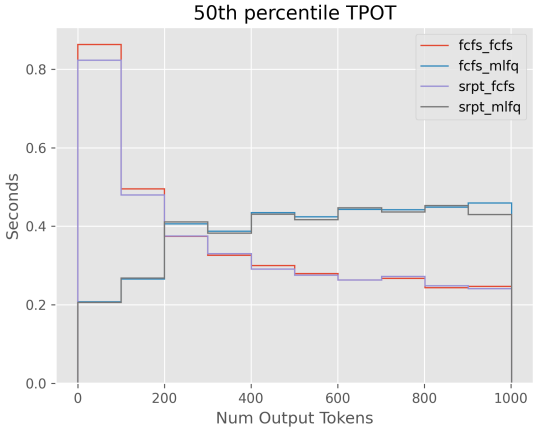

Figure 4: Median TPOT at 2 requests/second for various scheduler configurations.

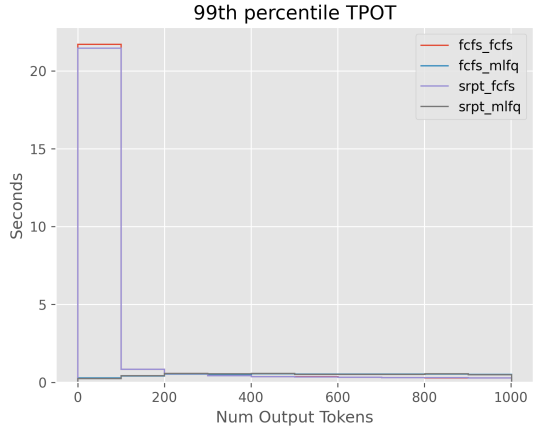

Figure 5: 99th Percentile TPOT at 2 requests/second for various scheduler configurations.

5.3 JCT Evaluation

We also analyze how the overall JCT for requests varies with the request arrival rate. We see in figure 6 that for short requests with the bottom 30% of generation lengths, using MLFQ for decode scheduling and SRPT for prefill scheduling significantly decreases both median and 99th percentile JCT latency. For the top 10% of requests, FCFS scheduling for both prefills and decodes achieves the lowest JCT latencies. This is expected since serving longer requests is purposefully delayed with SRPT and MLFQ scheduling to reduce latency for short requests. The starvation limit can also play a role in the inflated latency for long requests; with too high of a starvation threshold, some longer requests may not be scheduled soon enough with MLFQ decode scheduling. Tables 3 and 4 report JCT metrics for the whole distribution. At 1 request per second, using SRPT prefill and MLFQ decode scheduling marginally improves latencies. However, from 2 request per second onwards, we unfortunately see that due to the influence of longer requests, SRPT + MLFQ scheduling begins to perform worse than FCFS. Given the positive results for the short generation length slice of requests, we suspect that tuning parameters of the SRPT and MLFQ schedulers could improve JCT performance compared to FCFS for larger request arrival rates.

6 Limitations and Future Work

The biggest limitation with Prophet is the preemption overhead that MLFQ-style scheduling introduces in the Decoder, and is the root cause for both TPOT and JCT degradation for longer decode lengths.

We experimented with request length perception [13] to predict the decode lengths of each request to solve this problem. This would allow us to use a skip-join MLFQ scheduler similar to that of FastServe [11], reducing preemptions. However, we noticed that the instruction-tuned Llama3 8B did not predict response lengths accurately, often being very far off, which would hurt latency. We anticipate that larger models may be able to pro-

Scheduler	Median	90th %ile	99th %ile
fcfs_fcfs	21.573	47.490	74.790
fcfs_mlfq	20.868	46.191	80.580
srpt_mlfq	20.892	45.913	78.010

Table 3: JCT metrics at 1 requests/second for various scheduler configurations.

Scheduler	Median	90th %ile	99th %ile
fcfs_fcfs	79.134	134.478	192.171
fcfs_mlfq	86.730	200.230	279.851
srpt_mlfq	93.599	204.432	307.357

Table 4: JCT metrics at 2 requests/second for various scheduler configurations.

vide more accurate length predictions, but even some far-off predictions can have a large negative effect on head-of-line blocking, especially at the 99th percentile.

PagedAttention [5] is a more likely solution to preemption overhead. Instead of swapping out the entire KV cache of a request during preemption, each page of its KV cache can be paged out incrementally swapped out to main memory and pipelined to hide the latency while performing decode steps of the previous page. PagedAttention will also help with our high GPU memory usage, since our implementation currently stores all KV caches of requests in the scheduler in GPU memory to reduce latency.

Another limitation is that the starvation mechanism in our SRPT scheduler does not fully eliminate head-of-line blocking, especially at the 99th percentile. An alternative solution could potentially be a Multi-Level Queue (MLQ), but modified such that requests can promoted up one queue at a time rather than always to the top priority.

Lastly, our proof-of-concept only supports whole instances of Llama3 on each GPU. In practice, however, as future work, we will also need to optimize the model parallelism strategy since larger model will not fit on only one GPU. In this setting, Prefillers and Decoders may live on different nodes and we will need to consider bandwidth limitations in our model placement strategy as well. Fault tolerance would also be beneficial, especially when some nodes fail and Prefillers/Decoders go offline.

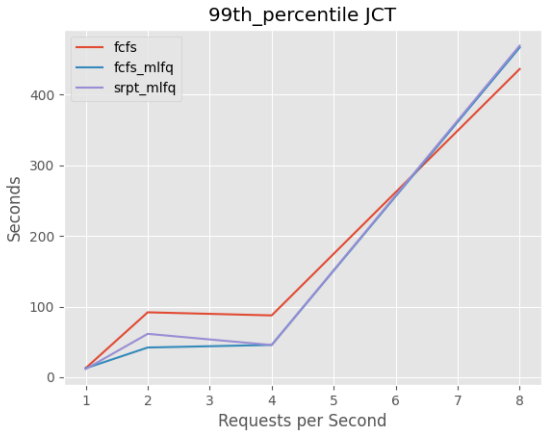

Figure 6: 99th Percentile JCT vs requests per second for bottom 30% of requests in generation length.

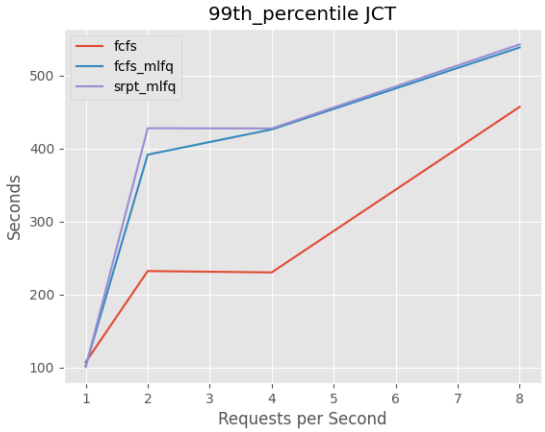

Figure 7: 99th Percentile JCT vs requests per second for top 10% of requests in generation length.

7 Conclusion

We present *Prophet* an LLM inference engine that is optimized for head-of-line blocking. We support iteration-level scheduling and disaggregate prefills and decodes. An SRPT scheduler is used for the prefill phase, an MLFQ scheduler with starvation prevention is used for the decode phase. Our experiments show that Prophet significantly reduces head-of-line blocking across TTFT, TPOT, and JCT metrics compared to techniques employed by start-of-the-art LLM inference systems.

References

[1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.

[2] AI@Meta. Llama 3 model card. 2024.

[3] anon8231489123. Sharegpt dataset card. 2023.

[4] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), page 335–344, New York, NY, USA, 1962. Association for Computing Machinery.

[5] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: a distributed framework for emerging ai applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 561–577, USA, 2018. USENIX Association.

[7] NVIDIA Corporation. Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution.

[8] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 606–624. Curan, 2023.

[9] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[11] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models, 2023.

[12] 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 *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[13] 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. *arXiv preprint arXiv:2305.13144*, 2023.

[14] 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. *arXiv preprint arXiv:2401.09670*, 2024.

8 Appendix: Strange results in Evaluation

Beyond the JCT results mentioned in the report, we saw two strange results in our evaluation. First, we saw that for TTFT, first come first serve schedulers actually had much lower TTFT than all other scheduler configurations combined. This is reflected in the median and 99th percentile latencies shown in figures 8 and 9. Second, we saw that SRPT prefill scheduling and FCFS decode scheduling had highly inflated latencies for both the bottom 30% and top 10% of requests in generation length. 99th percentile JCT results for these two request slices are show in figures 6 and 7.

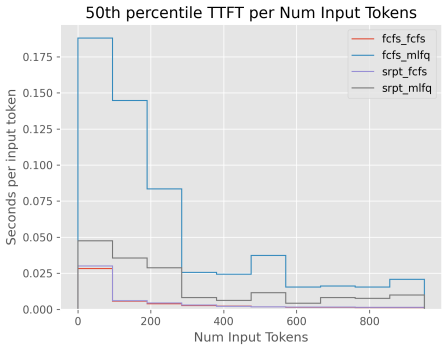

Figure 8: Median normalized TTFT at 2 requests per second. FCFS schedulers perform better than others.

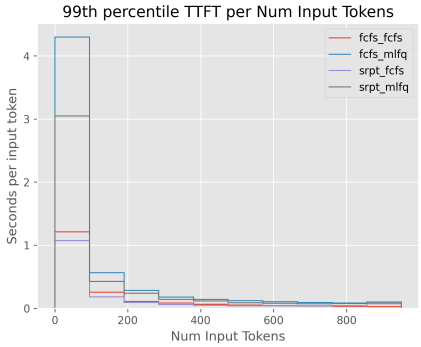

Figure 9: 99th percentile normalized TTFT at 2 requests per second. FCFS schedulers perform better than others.

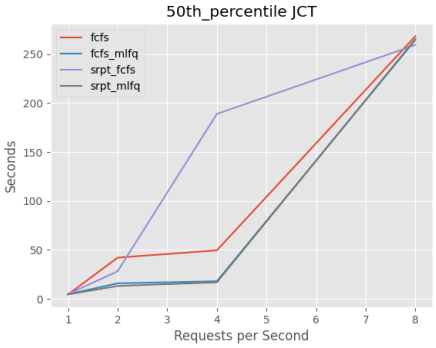

Figure 10: Median JCT metrics for bottom 30% of requests in generation length. SRPT+FCFS scheduler has strange behavior.

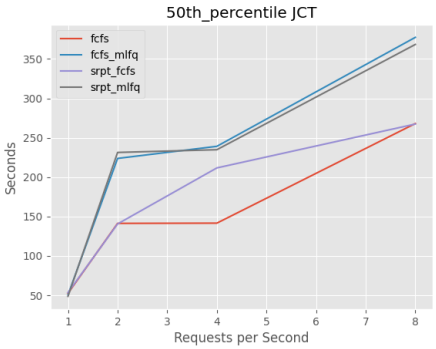

Figure 11: Median JCT metrics for top 10% of requests in generation length. SRPT+FCFS scheduler has strange behavior.