Paul Aldea (paldea), Michael Hwang (mihwang),
Ho Jung Kim (johnkimm), Jason Liang (jasonlia)

SUMMARY OF *ANDES: DEFINING AND ENHANCING QUALITY-OF-EXPERIENCE IN LLM-BASED TEXT STREAMING SERVICES*

Problem and Motivation

Surges in AI-driven applications (e.g. chatbots or real-time translation) have resulted in the increasing demand for scalable systems capable of handling high-work-generating requests from potentially even trivial prompts at an unprecedented rate. Such a unique workload has led to several systems-oriented optimizations and improvements for enhancing model throughputs and minimizing hardware expenditure (consider PagedAttention and SSMs). However, many advancements have commonly neglected the user experience in the entire exchange. Note that if a generative model used all available hardware resources to service a request in as short a time as possible, it is unlikely that the quality of experience (QoE) of the user is going to be drastically improved — there is a natural limitation for how much content a user could digest in a given timeframe. Furthermore, design decisions that are made for current LLM serving systems mainly aim to optimize "aggregated server-side metrics" by employing scheduling algorithms that are suboptimal to the user experience, such as first-come-first-serve (FCFS). As a result, the user experience drastically deteriorates under high server load (since token generation is likely to be significantly delayed) and also causes an inefficient use of hardware resources under lower workloads (since fast token generation is likely to make minimal difference to a client of the serving system). As such, the authors propose a QoE-aware scheduling mechanism to improve hardware utilization while minimizing the perceived user impact of such scheduling.

Related Works

The paper references works which aim at improving the management of hardware resources, though most solutions do not consider the role of the user QoE. For example, PagedAttention and iteration-level scheduling both optimize LLM-serving metrics, but fail to address the user-perceived slowdown of the time-to-first-token metric. Similarly, all other referenced works allow for greater model parallelism or hardware usage parallelism, but neglect the user QoE. As an example, a serving-system which employs a FCFS scheduling policy will very likely fail to deliver sufficient responsiveness to new requests (as these requests must be enqueued and are treated with lowest priority due to low age). On the user-side, this delay can feel like the application is sluggish and could result in the user canceling the given request or abandoning the usage of the application altogether. Further inquiries into existing works have led to little or no relevant solutions addressing the divide between QoE and server-oriented metrics, making Andes likely the first solution to improve hardware usage by leveraging human limitations.

Solution Overview

Andes proposes that a user's experience is mostly affected by the time it takes for the first token to generate (i.e. TTFT), followed by the token generation speed up to a maximum digestible rate by the user — the rate at which tokens should be expected to be served to the user for maximum satisfaction is known as the token delivery timeline. When the TDT matches the rate at which the user can digest the content, maximum QoE is obtained, while potentially saving computation done on by the LLM-serving system. More specifically, if a user is expected to observe the output of a generative model, the time it takes for the user to digest (i.e. engage with) a given generated output could be leveraged to reduce the speed at which content may need to be served and redirect resources to serve other requests. To leverage these properties, Andes proposes the following enhancements to existing systems:

- *Client-side generated-token buffer*: which stores generated tokens already served by the LLM to the client, with the hopes that system resources could be consequently diverted from the given request without impacting the user QoE. Generated tokens from the buffer can be shown to the user to best match the anticipated TDT established beforehand.
- *System resource preemption*: which allows for the system to preempt the serving of any given request to redistribute hardware resources based on a QoE-aware scheduling algorithm. In addition, an average preemption cap is also defined to ensure that system thrashing (repeated preemption with no useful work) can not occur, should system workload demands grow quickly.
- *Selective Triggering*: which allows for the system to determine when preemption needs to occur based on hardware exhaustion (such as memory limitations or token-generation latency). This also ensures that preemption does not occur unnecessarily if the overall system demand is relatively low.
- *Greedy Scheduling*: which allows for potentially suboptimal but routinely fast scheduling decisions to be made, limiting the amount of performance overhead incurred by scheduling the requests in addition to actually serving them.

Limitations

There are a few notable limitations to the proposed solution, namely:

1. The system was tested under a relatively limited preemption algorithm, which only allowed an average of one preemption for one request. As such, the significant speed-up noted by the paper in the TTFT metric under high load is actually well-expected and likely not representative of how the QoE of the user evolves over time. A given request is expected to experience a short TTFT, but could run into greater latency after preemption, should no other requests be available for preemption. Simply, a request can be initially quickly served, but once it is preempted, it has no guarantee that it will be resumed in a timely manner in order for the request to be fully serviced.
2. The evaluation of the QoE satisfaction metric was primarily derived on the average reading speeds of users in different demographic groups. However, it is likely (and

probable) that not all clients are users with fixed reading speeds. For example, serving a human user is likely to require less hardware resources than serving output to a software application which can easily digest many more tokens. As such, Andes provides no concrete way for distinguishing between the varying needs between varying users.

Future Research Directions

The paper doesn't discuss possible future research directions. The following are the possible research areas that our team came up with.

1. **Client-side Adaptation and Network Variability**
   This paper focuses primarily on server-side optimization. However, users often experience variations in network conditions, which can lead to inconsistencies in QoE despite optimal server performance. Andes can further incorporate client-side factors, like network condition, to dynamically adjust token delivery strategies. For example, it can communicate with client devices to adjust the token delivery speed, optimize packet sizes, or even prioritize responses based on client-side conditions.
2. **Andes for Video and Audio**
   Andes is designed specifically for text streaming services. However, similar QoE issues arise in other modalities, including video or audio, where factors like buffering, resolution, or sound quality impact the user experience. Future research could extend Andes' framework to work with other modalities. This would involve extending Andes' scheduling and resource allocation strategies to support multimedia QoE metrics such as resolution or sound quality.

Summary of Class Discussion

Q: For longer and more resource-hungry requests, especially those requiring previously generated responses, would that result in worse QoE?
A: If there are enough resources available, QoE would not be significantly affected. However, if resources are scarce, QoE will likely degrade for such requests.

Q: How much better is the optimal DP scheduling solution compared to the greedy algorithm?
A: The improvement from DP is not substantial enough considering its exponentially high overhead. The greedy algorithm offers a good balance between performance and efficiency.

Q: What happens during preemption, and how do we manage the KV cache of a preempted request?
A: A preemption happens when a request has already received some service and we put a pause on it to switch to working on a higher priority request — particularly when contended hardware resources are scarce**.** Need to store KV cache and change the context. Here we can offload KV cache to CPU memory or recompute, depending on available memory.

SUMMARY OF *FAIRNESS IN SERVING LARGE LANGUAGE MODELS*

Problem and Motivation

The key problem is to devise a method for an LLM provider to schedule requests from multiple users in a way that is “fair” and minimizes wasted resources. Here the term “fair” signifies that one user should not be able to limit the service to other users (e.g., by sending too many requests, by providing too many input tokens, or by sending requests that happen to generate too many output tokens) while still not overly restricting users with heavy requests as long as there are enough resources available. From a practical standpoint, this is an important problem because if left unresolved, it could lead to damaging user experiences (similarly as in the Andes paper) and wasting compute resources, power, and money. From research standpoint, making progress on this problem is essential since the more classical algorithms for fair queueing (e.g., in networking) have important shortcomings in the increasingly popular context of LLM serving, due to the challenges like the random output length and the algorithmic differences in the prefilling stage and the decoding stage (as discussed more in **Solution Overview**).

Related Works

The goal of achieving fairness when scheduling requests for limited resources is a relevant issue in many other domains including networking and operating systems. As such, the paper references previous works that tackle problems like link bandwidth allocation and CPU scheduling, such as fair queuing, weighted fair queuing, and start time fair queuing. However, as mentioned above, LLM serving presents additional problems (i.e., request lengths, input and output token costs, and effective capacity can all vary). The VTC algorithm summarized below strives to address these issues specific to the LLM serving. The paper also references works like Themsis and Pollux that deal with fairness in model training. This paper differs in that it tackles model serving instead, which has more to do with batching requests from multiple clients as opposed to assigning GPUs to clients. Note that the references include numerous works that present techniques for memory optimizations, model parallelism, and preemptive scheduling, but none address fairness among clients as considered in this paper.

Solution Overview

The paper’s solution is the algorithm called Virtual Token Counter (VTC). One key idea in the algorithm is to assign to each client a dynamically updating counter that monotonically increases as the server performs more service for that client (i.e., prefilling or decoding). Here the increase due to the prefilling phase tends to be weighed by a different constant of proportionality than the weight for the decoding phase; in fact, it is sensible to set the former to be smaller, as the prefilling is cheaper on average per token due to the fact that it analyzes the whole input prompts at once and thus enjoys better parallelization. In other words, we are thinking of a service as theoretically defined by taking an appropriate weighted sum of the number of tokens for the prefill phase and that for the decoding phase. One subtle but important point not addressed in this definition is the question of how exactly to set a counter of a new or returning client who does not have any active request in the queue. The answer can be found in the three motivating philosophies on “fairness”: First, if two clients have active requests, then we should not prefer one over the other; secondly, if one client has an active request and another client does not, then the former should be preferred or treated equally as the latter; and finally, the server cannot stall if the queue is nonempty. In particular, to satisfy the second philosophy, if one client with no active request in the queue submits a request, then the algorithm updates the client’s counter to the maximum counter among those in the queue, a process termed “counter lift.” Given the current queue of requests, the algorithm batches first-arrived requests from users with minimal counters and prefills them (while incrementing the counter as appropriate for prefilling). The prefilled requests are then decoded in a batch, followed by an analogous increase in the counter for corresponding users.

Limitations

One possible limitation is that the VTC algorithm does not take into account a user’s reading speed like Andes does, which may suggest the former is less considerate to optimizing the quality of user experience. For example, it may make more sense that a user with active requests is sometimes strictly prioritized over a newly joined user with a stream of shorter requests, as long as we can do this in a way such that the latter generates output tokens at or above the assumed reading speed of the user. After all, one could argue that a user would feel unfairly treated only if the delay in processing subsequent requests is slower than the user’s reading speed for the initial requests (as the user may be in the process of reading the output for the initial requests rather than waiting for later requests to be filled). Moreover, unlike Andes, the VTC algorithm does not allow any preemptions of requests, opting to modify priorities only by counter updates, even if there may be situations where preemption is sensible (such as when there is a user who only sends extremely token-heavy requests) in order to maximize the user experience quality as in Andes.

Future Research Directions

As noted above, preemption is not currently considered in the VTC algorithm. Adding a preemption mechanism would involve preempting requests when the difference in service crosses a threshold. This helps fix the problem of underestimating the fairness bound and is thus an exciting area for future research. In addition, the paper mentions adapting VTC for distributed systems. This would require a counter synchronization method as counters would be updated by different serving engines concurrently. Lastly, the paper brings up auto-scaling as an area for future research. Since VTC can handle changes in the number of GPUs, auto-scaling could address throughput degradation. As traffic changes, resources could be scaled. This would introduce auto-scaling related challenges like cost overhead and delays.

Summary of Class Discussion

Q: VTC has logic for updating counters. Is there overhead for running this algorithm, especially in the case where the update logic is not run on the local machine?
A: The overhead is minimal. The algorithm mainly selects the token counter and updates it, so it does not involve heavy computation or significant delays.

Q: Why is the absolute difference for accumulated service not "chasing tokens" in Scenario 2?
A: This is due to the granularity of the graph. The granularity limits the precision with which we can observe the changes in token accumulation over time.

Q: How does throughput compare for non-scheduling approaches like First-Come-First-Serve?
A: The throughput is similar between FCFS and the VTC approach when the Request Per Minute threshold gets larger. As the RPM limit increases, it effectively behaves like FCFS.

Q: Could there be overflow in the VTC counter since it only increments and never decreases?
A: In normal operations, overflow shouldn't be an issue unless the system runs for many years without resetting the counters. In such cases, the counters can simply be reset periodically to avoid overflow.

Q: When a new client arrives and chooses the minimum counter, what happens if a user hasn't requested service in a long time? Is it acceptable to pop or reset that user's counter?
A: It's fine not to pop or reset the user's counter. If the counter is set to 0, the system will prioritize this client the next time it makes a request.

Q: When weighting input and output tokens differently, how do you set the ratio?
A: It depends on the computational cost of processing each type of token. If output tokens are heavier to process, the output to input weight ratio might be set to 2:1. If the processing costs are similar, a 1:1 ratio might be appropriate.