Fairness in Serving Large Language Models

Ying Sheng[1,2] Shiyi Cao[1] Dacheng Li[1] Banghua Zhu[1] Zhuohan Li[1] Danyang Zhuo[3]
Joseph E. Gonzalez[1] Ion Stoica[1]

[1]UC Berkeley [2]Stanford University [3]Duke University

Abstract

High-demand LLM inference services (e.g., ChatGPT and BARD) support a wide range of requests from short chat conversations to long document reading. To ensure that all client requests are processed fairly, most major LLM inference services have request rate limits, to ensure that no client can dominate the request queue. However, this rudimentary notion of fairness also results in under-utilization of the resources and poor client experience when there is spare capacity. While there is a rich literature on fair scheduling, serving LLMs presents new challenges due to their unpredictable request lengths and their unique batching characteristics on parallel accelerators.

This paper introduces the definition of LLM serving fairness based on a cost function that accounts for the number of input and output tokens processed. To achieve fairness in serving, we propose a novel scheduling algorithm, the Virtual Token Counter (VTC), a fair scheduler based on the continuous batching mechanism. We prove a $2\times$ tight upper bound on the service difference between two backlogged clients, adhering to the requirement of work-conserving. Through extensive experiments, we demonstrate the superior performance of VTC in ensuring fairness, especially in contrast to other baseline methods, which exhibit shortcomings under various conditions.

1 Introduction

In a very short time, Large Language Models (LLMs), such as ChatGPT-4 Turbo [32], have been integrated into various application domains, e.g., programming assistants, customer support, document search, and chatbots. The core functionality rendered by LLM providers to these applications is serving their requests. In addition to the response accuracy, the request response time is a key metric that determines the quality of service being provided. Furthermore, LLM providers seek to utilize their resources efficiently so they can reduce costs and increase their competitiveness in the market.

Today's LLM serving systems [16, 20] typically use First-Come-First-Serve (FCFS) to schedule incoming requests. While simple, this scheduling discipline has several drawbacks. One such drawback is the lack of *isolation*: a client sending a disproportionate number of requests can negatively impact the service of all the other clients sharing the same server (i.e., slow down their requests or even cause timeouts) even when they send very little traffic. One solution to address this problem is to limit the incoming load of each client. Many of the existing LLM services do this today by imposing a request-per-minute (RPM) limit [33] for each client.

Unfortunately, RPM can lead to low resource utilization. A client sending requests at a high rate will be restricted even if the system is underutilized. This leads to wasted resources, an undesirable situation given the cost and the scarcity of GPUs. Thus, we want a solution that provides not only isolation (like RPM limit) but also high resource utilization.

This is a common problem in many other domains like networking and operating systems. The solution of choice to achieve both isolation and high resource utilization in those domains has been *fair queueing* [26]. Fair queueing ensures that each client will get their "fair share". In the simplest case, if there are n clients sharing the same resource, the fair share is at least $1/n$ of the resource, which means that each client gets at least $1/n$ of the resource. Furthermore, if some clients do not use their share, other clients with more demands can use it, hence leading to higher resource utilization.

In this paper, we apply fair sharing to the domain of LLM serving at the token granularity. We do it at the token rather than request granularity to avoid unfairness due to request heterogeneity. Consider two clients, client A sends requests of 2K tokens each (both input and output), and client B sends requests of 200 tokens each. Serving an equal number of requests for each client would be unfair to client B as her requests consume much fewer resources than client A's requests. This is similar to networking where fair queuing is typically

*Part of the work was done when Ying was visiting UC Berkeley.

applied to the bit granularity, rather than packet granularity.

Despite these similarities, we cannot directly use the algorithms developed for networking and operating systems, as LLM serving has several unique characteristics. First, the request output lengths are unknown in advance. In contrast, in networking, the packet lengths are known before the packet is scheduled. Second, the cost of each token can vary. For instance, the cost of processing an input (prompt) token is typically lower than that of an output token, because input tokens processing is parallelizable. In contrast, the cost of sending a bit or the cost of a CPU time slice are the same irrespective of the workload. Third, the *effective* capacity of an LLM server (i.e., processing rate expressed in token/sec when the request queue is non-empty) can vary over time. For example, longer input sequences take more memory. This limits the number of batched parallel requests during generation, leading to GPU under-utilization and a lower processing rate. In contrast, the network or CPU capacity is assumed to be fixed.

In this paper, we discuss the factors that need to be considered when defining fairness in the context of LLM serving. We show how different definitions can be incorporated into a configurable cost function (Section 3). We then present a fair scheduling algorithm called *Virtual Token Counter (VTC)* that can be easily adapted for different cost functions. At a high level, VTC tracks the services received for each client and will prioritize the ones with the least services received, with a counter lift each time a client is new to the queue. It updates the counters at a token-level granularity on the fly, which addresses the unknown length issue. VTC integrates seamlessly with current LLM serving batching techniques (Section 2.1), and its scheduling mechanism does not depend on the server's capacity, overcoming the problem of the dynamically fluctuating server capacity. We also provide theoretical bounds of fairness for VTC in Section 4.1. The system architecture of VTC implementation in illustrated in Figure 1.

In summary, this paper makes the following contributions:

- This is the first work to discuss the fair serving of Large Language Models. We identify its unique challenges and give the definition of LLM serving fairness (Section 3).
- We propose a simple yet effective fair-serving algorithm called VTC. We provide rigorous proofs for VTC on fairness guarantee, which gives fairness bound within $2\times$ of the optimal bound (Section 4).
- We conduct in-depth evaluations on our proposed algorithm VTC. Results confirm that our proposed algorithms are fair and work-conserving, while other alternatives can fail in different settings (Section 5).

2 Background

In this section, we first introduce how an LLM serving system operates. Then we describe existing methods for ensuring fairness in LLM serving.

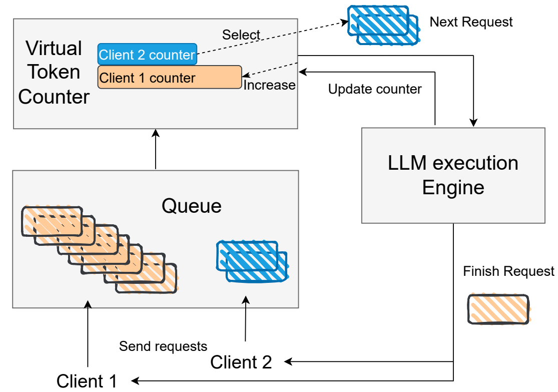


Figure 1: Serving architecture with Virtual Token Counter (VTC), illustrated with two clients. VTC maintains a queue of requests and keeps track of tokens served for each client. In each iteration of the LLM execution engine, some tokens from some clients are generated. The counters of these clients are correspondingly updated. When the condition of adding new requests is satisfied (e.g. memory is released when some other requests finish), VTC will be invoked to choose the requests to be added. VTC achieves fairness by prioritizing clients with the lowest counter (Section 4.1).

2.1 Large Language Models Serving

LLM serving with a single request First, a request contains information about its arrival time (a), input tokens (x), and its associated client (u). Formally, we represent a request using a three-tuple (a, x, u). The system generates output tokens based on the input tokens. For instance, the input tokens can be an incomplete sentence, and the system generates the rest of the sentence [31].

The generation procedure consists of two stages: the initial **prefilling** stage, and the **decoding** stage [35]. Mathematically, x is a sequence of tokens $(x_1, x_2, \ldots, x_n)$. In the prefilling stage, the LLM computes the probability of the first new tokens: $P(x_{n+1}|x_1, \ldots, x_n)$. In the decoding stage, the system *autoregressively* generates a new token. At time t ($t \geq 1$), the process is written as:

$$P(x_{n+t+1}|x_1, \ldots, x_{n+t})$$

The decoding stage ends when the LLM generates a special end-of-sentence (EOS) token or the number of generated tokens reaches a pre-defined maximal length.

LLM serving with multiple requests In the online serving scenarios, multiple clients submit requests to the serving system. To process these requests, the system maintains two concurrent streams: A monitoring stream adds requests to a waiting queue; an execution stream selects and executes request(s) from the waiting queue.

Naively, the execution stream can choose to execute requests one by one. However, this is highly GPU inefficient due to various natures of the LLM generation procedure. For instance, the decoding steps must be carried out sequentially where the arithmetic intensity is relatively low in a single step. Contemporary serving systems usually perform batching that executes multiple requests concurrently to maximize the system throughput. The most widely used approach in LLM serving is continuous batching [46]. Algorithm 1 shows the pseudocode for continuous batching.[2] The monitoring stream enqueues requests to a waiting queue. The execution stream performs a check on whether there are finished requests at the end of each decoding step. If there are, the system removes these requests and adds new requests from the queue.

Fairness with continuous batching We can naturally integrate fairness policies into the continuous batching algorithm, by designing a fair `select_new_requests()` function in Algorithm 1. Intuitively, the execution stream should keep track of how much service a particular client has received, and prioritize clients that haven't received much service in the next selection. We formally define fairness in the LLM serving context in Section 3 and design a method with theoretical guarantee in Section 4.

We adopt a continuous batching scheme where a request only leaves the batch when it generates an EOS token or reaches the pre-defined maximum number of generated tokens (i.e., no preemption). This paper focuses on how to integrate fair scheduling with continuous batching, and we leave an investigation on preemption as an orthogonal future work.

Algorithm 1 LLM serving with Continuous batching

Initialize current batch $B \leftarrow \emptyset$, waiting queue $Q \leftarrow \emptyset$
▷ `with monitoring stream:`
while True **do**
    if new request r arrived **then**
        $Q \leftarrow Q + r$
▷ `with execution stream:`
while True **do**
    if can_add_new_request() **then**
        $B_{new} \leftarrow$ **select_new_requests(Q)**
        prefill(B_{new})
        $B \leftarrow B + B_{new}$
    decode(B)
    $B \leftarrow$ filter_finished_requests(B)

2.2 Existing Fairness Approaches

Fairness is a key metric of interest in computer systems that provide service to multiple concurrent clients [3]. A *fair* LLM serving system should protect clients from a misbehaving client who may try to overload the serving system by submitting too many requests.

RPM Limit Per Client As a common practice of API management (e.x. [33]), specific rate limits are established for each client's API usage to prevent potential abuse or misuse of the API and ensure equitable access for all clients. This limitation is on the metric request-per-minute (RPM). Once a client reaches the RPM limit, the client is only allowed to submit more requests in the next time window. However, it's important to note that while these limits are effective in managing resource allocation during periods of high demand, they may not be *work-conserving* when the number of active clients is low. In such scenarios, the system's capacity might be underutilized, as the imposed limits prevent the full exploitation of available resources.

Fair Queueing [26] The fairness problem has been extensively studied in the past for traditional compute resources, such as CPU cycles and network bandwidth. Fair queuing and its variants (e.g., Weighted Fair Queuing (WFQ) [8], Self-clocked Fair Queueing [12], and Start-time Fair Queueing (SFQ) [14]) have been proposed to achieve the fair allocation of link bandwidth in packet-switching networks.

In the traditional packet-switching network, a *flow* f is referred to as a sequence of packets $p_f^0, p_f^1, \dots p_f^n$ transmitted by a source. Each packet p_f^j is of length l_f^j. A flow is *backlogged* during the time interval $[t_1, t_2)$ if it has one or more outstanding packets waiting in the queue at any time $t \in [t_1, t_2)$.

Virtually all fair queueing algorithms maintain a system *virtual time*, $v(t)$, which intuitively measures the service received by a continuously backlogged flow in terms of bits forwarded. Each packet, p is associated two tags: *Start* tag $S(p)$ and a *Finish* tag $F(p) = S(p) + l_p$. The Start tag (a.k.a. packet's virtual starting time) is computed based on both the system virtual time and the Finish tag (a.k.a. packet's virtual finishing time). These algorithms schedule packets in the ascending order of either the Finish or Start tags.

In networking, fairness is simply defined as follows: for any two flows, f and g, that are backlogged during time interval $[t_1, t_2)$, we have

$$\left| W_f(t_1, t_2) - W_g(t_1, t_2) \right| \le U(f, g), \tag{1}$$

where $W_f(t_1, t_2)$ and $W_g(t_1, t_2)$ denote the service received in bits by flow f and g, respectively, during interval $[t_1, t_2)$, and $U(f, g)$ is a function of the properties of flows f and g (e.g., maximum packet length) and the system (e.g., link capacity). Intuitively, for packets-switching networks, the allocation of a link bandwidth is fair if, for any time interval during which two flows are backlogged, each of these flows receives approximately the same service in terms of the number of bits being forwarded during that interval. A scheduling algorithm is said

[2] For a simple presentation, we consider the continuous batching from TGI [17] rather than the original proposed iteration-level scheduling in Orca [46].

to be *work-conserving* if a link always forwards packets when the queue is not empty [19].

There exists a distinct strand of research [2, 4, 43] focusing on the fair scheduling of preemptible tasks (e.g., CPU scheduling). However, our setting for LLM serving aligns more closely with the packet scheduling problem, and introduces new challenges such as unpredictable output length, cost disparity between input and output tokens, and variable token-rate servers.

2.3 Challenges

There are several unique challenges in LLM serving that prevent a direct application of fair-queuing-like algorithms. The first challenge is that the definition of fairness in the context of LLM serving is unexplored, and likely very different than that discussed in fair-queuing literature.

Traditional fairness is defined by measuring the cost of requests, which is usually a fixed value that is easy to estimate in either network or operating systems. For example, in networking, requests correspond to packets, and the cost is usually the number of bits of a packet. However, in LLM generations, how to define the cost of a request is not obvious. The cost per token can vary. Especially, processing an input (prompt) token is typically less expensive than processing an output token, as input tokens are processed in parallel while output tokens must be generated sequentially. Batching the output tokens from different requests can parallelize the fully connected layers but is still slower than processing input tokens for the attention layers.

Additionally, in LLM serving, the server has variable token-rate capacity, although the memory allocated for a batch is constant. Firstly, even if the request queue is not empty, we are not guaranteed that each batch is full. This is because we need to preserve spaces for future generated tokens, and also because the tokens added to the batch are not at the token but the request granularity. Secondly, the number of tokens processed highly depends on the requests' arrival patterns because of the continuous batching mechanism (Section 2.1). Furthermore, the capacity depends on the mix between input and output tokens of existing requests. If all requests have long past tokens, then the capacity is likely to be low (See Figure 2). Then there is no way to define a fixed amount of equal share.

The second challenge is the characteristic of unknown output length before finishing a request. This prevents a direct adaptation of classical algorithms like SFQ and Deficit Round Robin (DRR) [41] into the LLM serving. SFQ-style algorithms can provide good bounds in fairness by setting the Start and Finish tags through virtual time, as introduced in Section 2.2. However, computing Start and Finish tags requires knowing the request length in advance. DRR performs round-robin scheduling with a "deficit counter" mechanism to achieve fair scheduling of packets of variable length. In DRR, each client is assigned a specific quantum of service. It

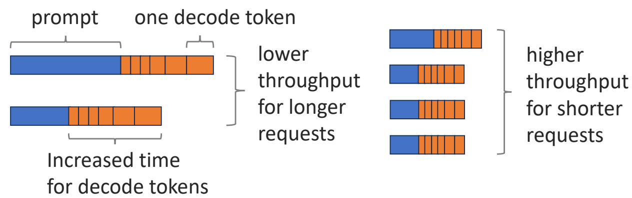


Figure 2: An illustration of how request length can affect the cost and server capacity in terms of throughput. The visualized length is not precise but for illustration purposes only.

tracks the "deficit" of service for each client to ensure fairness over time. During each round, the scheduler allows each client to dispatch as many requests as possible, provided that the total length of these requests does not exceed the sum of the client's assigned quantum for that round and any accumulated deficit from previous rounds. Without knowing the length in advance, DRR cannot determine how many jobs can be scheduled within the quantum.

We will give the definition for LLM serving fairness in Section 3, and give an algorithm to achieve the LLM serving fairness in Section 4.

3 Definition of Fairness in LLM Serving

As discussed in Section 2.3, how to define the cost of a request in LLM serving is not obvious. In this section, we discuss how to define the cost for a request, and how to measure the service a client has received (Section 3.1). After defining the measurement of service, we can define fairness among clients in Section 3.2.

3.1 Measurement of Service

In this subsection, we will discuss the measurement of the service a client has received. Specifically, we define $W_f(t_1, t_2)$ and $W_g(t_1, t_2)$ from Equation (1) in the context of LLM serving. We omit the subscript, $W(t_1, t_2)$, when the client is clear from the context or irrelevant.

Number of tokens A straightforward way to measure the service provided to a client is by summing the number of input tokens have been processed and the number of output tokens have been generated so far, i.e., $W(t_1, t_2) = n_{input}(t_1, t_2) + n_{output}(t_1, t_2)$.

Number of FLOPs Alternatively, one can measure the total FLOPs used in each stage, i.e., $W(t_1, t_2) = FLOP_{input}(t_1, t_2) + FLOP_{output}(t_1, t_2)$. This can be more precise because it captures the difference among tokens in attention computation, where tokens with longer prefixes require more computation.

However, both of these formulations cannot accurately reflect the actual LLM serving cost: The computation of the tokens at the prefill stage can be parallelized and achieve high GPU utilization. However, at the generation stage, we can only generate tokens one by one, as each token depends on all previous tokens as described in Section 2.1.

Weighted number of tokens To better reflect the actual LLM serving cost, a more accurate measure should capture the difference in costs of the prefilling and generation phases. One simple way to implement this idea is by using a weighted combination of the prefilling (input) tokens and decoding (output) tokens, inspired by the pricing mechanism used in OpenAI's API[3]. Formally, let w_p be the weight of input tokens and w_q be the weight of output tokens. Then, we have $W(t_1,t_2) = w_p \cdot n_p + w_q \cdot n_q$, where n_p, n_q are the number of processed input and output tokens, respectively. Following OpenAI pricing, we set $w_p = 1$ and $w_q = 2$. Due to its simplicity, we will use this measure in our analysis and evaluation.

Customized, unified representation. The definition of fairness in LLM serving can also be extended to other aspects, such as the weighted number of FLOPs or a more sophisticated method introduced in [27] that uses piecewise linear functions for the number of input and output tokens. The important point to note here is that we need to separate the cost for input and output tokens. Generally, the service can be represented as a function of the number of input and output tokens (l_{int}, l_{out}, respectively). Let $h(l_{in}, l_{out})$ be the cost function that is monotonically increasing according to l_{in} and l_{out}. Our method can easily accommodate to different h (Section 4.2).

3.2 Fairness in LLM Serving

In this paper, we apply fair sharing to the domain of LLM serving to provide performance isolation across multiple clients sharing the same LLM server. In particular, we employ the classic formulation of max-min fairness [3], which computes a *fair share* for the clients sharing a given server. In a nutshell, given the metric of service fairness, if a client sends requests at no more than its fair share, all its requests are served. In contrast, if a client sends requests at more than its fair share, its excess requests will be delayed or even dropped. As a result, a misbehaving client cannot deny the service to other clients, no matter how many requests it sends. To achieve max-min fairness, an idealized serving system follows desirable properties as below:

1. **Backlogged clients** Any two clients f, g that are continuously backlogged during a given time interval $[t_1, t_2)$ should receive the same service during this interval, i.e. $W_f(t_1,t_2) = W_g(t_1,t_2)$.
2. **Non-backlogged clients** Client f that is continuously backlogged during time interval $[t_1,t_2)$ should not receive less service than another client, g, that is not continuously backlogged during the same time interval, i.e., $W_f(t_1,t_2) \geq W_g(t_1,t_2)$.
3. **Work-conservation** As long as there are requests in the queue, the server should not be idle.

The first property means that two clients sending requests at more than their fair share will get the same service, regardless of the discrepancy between their sending rates. The second property says that a client sending requests at a higher service rate will not get less service than a client sending at a lower service rate. Basically, the first two properties say that a misbehaving client is contained (i.e., doesn't receive more service than other backlogged clients), not punished (i.e., doesn't receive less service than other non-backlogged clients). Finally, the work conserving property aims to maximize the utilization, addressing a key weakness of the RPM-based solutions.

The three properties above assume an idealized fair serving system. A practical system will invariably approximate these properties. In general, the best we can achieve is deriving bounds that are independent of the length of the time interval, e.g., in the first property, the difference between $W_f(t_1,t_2)$ and $W_g(t_1,t_2)$ is bounded by a value that is independent of $t_2 - t_1$. We give the formal guarantees provided by our method in Section 4.1.

4 Achieving Fairness

In this section, we present our algorithm VTC with proved fairness properties in Section 4.1, and show its generalization for customized service measurement in Section 4.2.

4.1 Virtual Token Counter (VTC)

Based on insights from prior discussions, we've identified three key challenges inherent in Large Language Model (LLM) serving that hinder the effective adaptation of existing algorithms for delivering approximately fair LLM service: (1) The cost per token can vary, i.e., the service depends on both the number of input and output tokens, (2) The server has variable token-rate capacity, and (3) the output length is unpredictable.

To address these three challenges, we propose the Virtual Token Counter (VTC), a mechanism for achieving fair sharing in LLM Serving (Algorithm 2). To quantify the service received by a client we use the weighted number of tokens metric, as described in Section 3.1. We discuss the generalization to other metrics in Section 4.2.

Intuitively, VTC tracks the services received for each client and will prioritize the ones with the least services received, with a counter lift each time a client is new to the queue. The *counter lift* is needed to fill the gap created by a low load period of the client, so that it will not be unfairly served more in the future. In other words, the credits for a client are utilized immediately and cannot be carried over or accumulated. The virtual counters are updated each time a new token is generated, which can reflect the services received instantly. This

这段没看懂啥意思

[3]https://openai.com/pricing

operates at the token-level granularity, and thus addresses the unknown length issue. VTC can be easily integrated into the continuous batching mechanism, and its scheduling mechanism does not depend on the server's capacity, overcoming the problem of variable token-rate capacity.

Algorithm 2 shows how VTC can be implemented in the continuous batching framework described in Section 2.1. It maintains a virtual counter for each client, denoted as $\{c_i\}$. The counters are initialized as 0 (line 2). The program runs with two parallel streams.

The monitoring stream listens to the incoming requests, described in lines 5-14. The new request will be added to the waiting queue Q immediately. If the new request is the only request in Q for its sender client, a counter lift (lines 8-13) will happen. Because this client could have been underloaded before, its counter could be smaller than the other active counters. However, since the credits cannot be carried over, we need to lift it to the same level as other active counters, thus maintaining fairness among this client and others. Lines 9-10 address the scenario where the entire system was in an idle state. We do not reset all the counters to avoid nullifying a previously accumulated deficit upon a system restart.

The execution stream is the control loop of an execution engine that implements continuous batching. Line 17 controls the frequency of adding a minibatch B_{new} of new requests into the running batch B. Commonly, the server will add a new minibatch after several decoding steps. The minibatch B_{new} is constructed by iteratively selecting the request from the client with the smallest virtual counter (lines 20-25). The counters will be updated when adding new requests according to the service invoked by the input tokens (line 23). After each decode step (line 28), $\{c_i\}$ will be updated immediately according to the service invoked by the newly generated output tokens (line 29).

The VTC algorithm is (mostly) work-conserving because it only manipulates the dispatch order and does not reject a request if it can fit in the batch.

Algorithm 2 Virtual Token Counter (VTC)

Input: request trace, input token weight w_p, output token weight w_q, upper bound from Equation (2) denoted as U.

1: let current batch $B \leftarrow \emptyset$
2: let $c_i \leftarrow 0$ for all client i
3: let Q denote the waiting queue, which is dynamically changing.
4: ▷ `with monitoring stream:`
5: **while** True **do**
6:     **if** new request r from client u arrived **then**
7:         **if** not $\exists r' \in Q, client(r') = u$ **then**
8:             **if** $Q = \emptyset$ **then**
9:                 let $l \leftarrow$ the last client left Q
10:                 $c_u \leftarrow \max\{c_u, c_l\}$
11:             **else**
12:                 $P \leftarrow \{i \mid \exists r' \in Q, client(r') = i\}$
13:                 $c_u \leftarrow \max\{c_u, \min\{c_i \mid i \in P\}\}$
14:         $Q \leftarrow Q + r$
15: ▷ `with execution stream:`
16: **while** True **do**
17:     **if** can_add_new_request() **then**
18:         $B_{new} \leftarrow \emptyset$
19:         **while** True **do**
20:             let $k \leftarrow \arg\min_{i \in \{client(r) \mid r \in Q\}} c_i$
21:             let r be the earliest request in Q from k.
22:             **if** r cannot fit in the memory **then** Break
23:             $c_k \leftarrow c_k + w_p \cdot input\_length(r)$
24:             $B_{new} \leftarrow B_{new} + r$
25:             $Q \leftarrow Q - r$
26:         forward_prefill(B_{new})
27:         $B \leftarrow B + B_{new}$
28:     forward_decode(B)
29:     $c_i \leftarrow c_i + w_q \cdot |\{r \mid client(r) = i, r \in B\}|$
30:     $B \leftarrow$ filter_finished_requests(B)

4.1.1 Fairness for backlogged clients in VTC

In this subsection, we provide the theoretical guarantee for fairness among overloaded clients in VTC. More precisely, the overload of a client is reflected by its backlog, which can be formally defined as follows. Intuitively, a client being backlogged means its requests are queued up.

Definition 4.1 (Backlog)**.** A client f is backlogged during time interval $[t_1, t_2)$, if at any time $t \in [t_1, t_2)$, f has a request that is waiting in the queue.

We adapt the traditional definition of fairness for backlogged clients in the network to our scenario as follows.

Definition 4.2 (Fairness adapted from [13])**.** Let $W_f(t_1, t_2)$ be the aggregated service received by client f in the interval $[t_1, t_2)$. A schedule is fair w.r.t. δ, if for any clients f and g, for all intervals $[t_1, t_2)$ in which clients f and g are backlogged, we have $|W_f(t_1, t_2) - W_g(t_1, t_2)| \le \delta$.

In the rest of the paper, as in Algorithm 2, we let Q denote the set of requests in the waiting queue. We abuse the notation of $i \in Q$ for a client i to indicate there exists $r \in Q$, such that r is a request from client i. Let L_{input} and L_{output} be the maximum number of input and output tokens in a request. Let M be the maximum number of tokens of a client that can be fitted in a running batch. Next, we start with some lemmas to prove a bound for Definition 4.2 in Theorem 4.4. The missing proofs are in Appendix A.

Lemma 4.3. *The following invariant holds at any time in Algorithm 2 when $Q \neq \emptyset$:*

$$\max_{i \in Q} c_i - \min_{i \in Q} c_i \le \max(w_p \cdot L_{input}, w_q \cdot M) \qquad (2)$$

Theorem 4.4 (Fairness for overloaded clients)**.** *For any clients f and g, for any time interval $[t_1,t_2)$ in which f and g are backlogged, Algorithm 2 guarantees* [4]

$$|W_f(t_1,t_2) - W_g(t_1,t_2)| \le 2\max(w_p \cdot L_{input}, w_q \cdot M).$$

Proof. For any f, if f is backlogged during time t_1 to t_2, we have $W_f(t_1,t_2) = c_f^{(t_2)} - c_f^{(t_1)}$. This is because the line 7 will not be reached for client f during t_1 to t_2, and the c_f keeps increasing during t_1 to t_2 by adding w_p product the number of served input tokens and w_q product the number of served output tokens. By Lemma 4.3, from Equation (2), we have

$$\begin{aligned}|W_f(t_1,t_2) - W_g(t_1,t_2)| &\le |c_f^{(t_1)} - c_g^{(t_1)}| + |c_f^{(t_2)} - c_g^{(t_2)}| \\ &\le 2\max(w_p \cdot L_{input}, w_q \cdot M)\end{aligned}$$

□

Remark 4.5. An empirical illustration of this theorem can be found in Figure 3a, where the difference between services received by backlogged clients is bounded, regardless of how long they have been backlogged.

Remark 4.6. Line 13 can be modified to take any value between $\min\{c_i | \exists r' \in Q, client(r') = i\}$ and $\max\{c_i | \exists r' \in Q, client(r') = i\}$. The proof of Theorem 4.4 should still hold.

Remark 4.7. To tighten the bound in Theorem 4.4, we can restrict the memory usage for each client in the running batch. This might compromise the work-conserving property, as we will demonstrate in Theorem 4.8. Therefore, there is a *trade-off* between achieving a better fairness bound and maintaining work conservation.

We also prove in the next theorem that the bound in Theorem 4.4 is tight within a factor of 2 for a family of work-conserving schedulers. We say a scheduler is work-conserving if it stops adding requests to a partially-filled minibatch (line 21 in Algorithm 2) only when it runs out of memory[5] but not for fairness reasons.

Theorem 4.8. *For any work-conserving schedule without preemption, there exists some query arrival sequence such that for client f,g and a time period t_1,t_2, such that*

$$|W_f(t_1,t_2) - W_g(t_1,t_2)| \ge w_q \cdot M,$$

where clients f,g are backlogged during the time $[t_1,t_2)$.

As we mentioned before, output tokens are more expensive than input tokens, so normally we have $w_q > w_p$. Therefore the right-hand side of the inequality in Theorem 4.4 is $2w_q \cdot M$, which is $2\times$ of the lower bound in Theorem 4.8.

[4]The service of a served request incurred by pre-filling (service for input tokens) is counted at the time when the request is added to the running batch (line 23 in Algorithm 2), rather than the time when prefill is finished. This is because we want to count the input tokens immediately to avoid selecting all the same k at line 20 in Algorithm 2 for B_{new}.

[5]Different implementation may have different criteria of "not enough memory". This can only be achieved heuristically because the number of output tokens is unknown before it finishes.

4.1.2 Fairness for non-backlogged clients in VTC

In this subsection, we discuss item 2 in Section 3.2. A backlogged client will not receive less service than another client. This can be reflected in the following theorem. The missing proofs are in Appendix A.

Theorem 4.9. *If a client f is backlogged during time interval $[t_1,t_2)$, for any client g, there is*

$$W_f(t_1,t_2) \ge W_g(t_1,t_2) - 4U.$$

Here U is the upper bound from Equation (2)*.*

In addition to that, clients who send requests constantly less than their share should have their requests serviced nearly instantly. This property intuitively can be implied by the first item in Section 3.2, as if a low-rate client cannot be served on time, it becomes backlogged, which requires the same level of service with backlogged clients. We formally prove this property to offer a fairness assurance for clients who are not overloaded in Theorem 4.13. This intuitively acts as a safeguard against misbehaving clients [7].

We start with Theorem 4.11 discussing the aspect of latency bounds. Intuitively, if a client is not backlogged and has no requests running, the next request from it will be processed within a latency bound that is independent of the request rate of other clients.

Definition 4.10. Assume there are n active clients during $[t_1, t_2)$, and the server capacity at time $t \in [t_1,t_2)$ is defined as $S(t)$, where

$$\int_{t_1}^{t_2} S(t)\,dt = \sum_{i=1}^{n} W_i(t_1,t_2)$$

Because the server capacity is always positive and bounded, there exists $a,b \in \mathbf{R}^+$ such that $\forall$ t, $a < S(t) \le b$.

Theorem 4.11. *Let $A(r)$ and $D(r)$ denote the arrival time and dispatch time of a request r. Assume there are in total n clients, $\forall t_1,t_2$, if at t_1, a client f is not backlogged and has no requests in the running batch, then the next request r_f with $t_1 < A(r_f) < t_2$ will have its response time bounded:*

$$D(r_f) - A(r_f) \le 2\cdot(n-1)\cdot\frac{\max(w_p \cdot L_{input}, w_q \cdot M)}{a} \quad (3)$$

Here a is the lower bound of the capacity in Definition 4.10.

Remark 4.12. The bound in Theorem 4.11 is irrelevant to the request rate of others, giving an upper bound for latency against ill-behavior clients.

The above is about one request not getting delayed. The following theorem shows that during time period $[t_1,t_2)$, if there are n active clients sending requests, and client f is sending requests with a rate constantly less than $1/n$ of the server's capacity (with some constant gap), client f should have all its requests been served.

Theorem 4.13. *(Fairness for non-overloaded clients) For any time interval $[t_1, t_2)$, we claim the following.*

Assume a client f is not backlogged at time t_1 and for any time interval $[t, t_2), t_1 \le t < t_2$, f has requested services less than $\frac{T(t,t_2)}{n(t,t_2)} - 5U$, where $T(t, t_2)$ is the total services received for all clients during the interval $[t, t_2)$, $n(t, t_2)$ is the number of clients that have requested services during the interval, and U is the upper bound from Equation (2).

Then, all of the services requested from f during the interval $[t_1, t_2)$ will be dispatched.

4.2 Adapt to Different Fairness Criteria

Algorithm 2 is designed for fairness with the service function $W(t_1, t_2)$ as a linear combination of the number of processed input tokens and the number of generated tokens. For a different definition of $W(t_1, t_2)$, Algorithm 2 can be easily modified to update the counter according to the other definitions described in Section 3.1.

For example, let n_{in} and n_{out} be the number of processed input tokens and generated tokens, respectively. Assume we are pursuing fairness with $h(n_{in}, n_{out})$ as the service measurement for some function h. Line 23 will be changed to

$$c_k \leftarrow c_k + h(l_{in}^r, 0).$$

Line 29 will be changed to

$$c_i \leftarrow c_i + \sum_{r|client(r)=i, r\in B} \left(h(l_{in}^r, n_{out}^r) - h(l_{in}^r, n_{out}^r - 1)\right).$$

Here l_{in}^r denotes the input length of request r, and n_{out}^r denotes the number of generated tokens of request r.

The fairness bound will also be changed according to $h(n_{in}, n_{out})$. Under the assumption that output tokens are more expensive than input tokens, the bound will become the maximum value of aggregated $h(\cdot,\cdot)$ for a set of requests that can be fitted in one running batch.

5 Evaluations

In this section, we evaluate VTC against other alternatives under different workloads. The results confirm the fairness properties introduced in Section 3 of VTC, and show that all other alternatives will fail in at least one workload.

5.1 Setup

Implementation We implement our VTC and other baseline schedulers in S-LoRA [39], a system that serves a large amount of LoRA adapters concurrently. Its backbone is a general serving system adapted from LightLLM [25]. It includes the implementation of continuous batching [46] and PagedAttention [20][6]. Our VTC scheduler is built on top of those two techniques. Fairness can be considered among general clients, and our experiments are done in this way. But we would like to note that fairness also could be taken into consideration among adapters, especially under the scenario of personalization that uses one adapter per customer, which originally motivated this paper.

Baselines In this section, we benchmark VTC and the baselines as below:

- First Come First Serve (FCFS): In the First-Come-First-Serve method, requests are handled strictly in the order they are received, irrespective of the requesting client. This is the default scheduling strategy in many prevalent LLM serving systems, including vLLM [20] and Huggingface TGI [16].
- Request per minute (RPM): This method limits the maximum number of requests that a client can make to the server within a one-minute timeframe. The definition of service corresponds to Section 3. When a client exceeds this limit, subsequent requests are blocked until the limit resets at the start of the next minute.
- Least Counter First (LCF): This is a variant of VTC without the counter lift component. Each client will maintain a counter for the service it received so far. The request from the client with the smallest counter will be scheduled each time.

Synthetic Workload We run Llama-2-7b on A10G (24GB), using the memory pool for the KV cache with size 10000[7]. We use various workloads to demonstrate different aspects of fairness, and compare VTC with other baselines. The detailed results are in Section 5.2. We start with synthetic workloads to give a clear message for fairness properties.

Real Workload To validate the effectiveness of VTC in more complex real-world scenarios, we also experiment with VTC and other baselines under workloads constructed from the trace log of LMSYS Chatbot Arena [48, 49], which is an LLM serving platform for real-world clients.

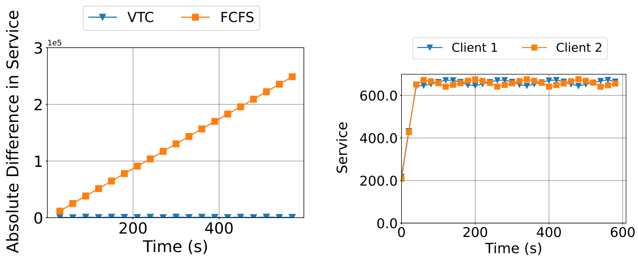
Ablation Study In the ablation study, to evaluate the impact of different memory pool sizes and request lengths on the scheduling fairness, we run Llama-2-13b on A100 (80GB) with a memory pool of size 35000 and 65000 respectively. For each memory pool size, we evaluate the absolute difference in the accumulated service of two clients.

5.2 Results on Synthetic Workloads

We design a set of experiments to visualize the fairness properties of VTC. We start with synthetic traces to show plots

[6] with block size equals 1.

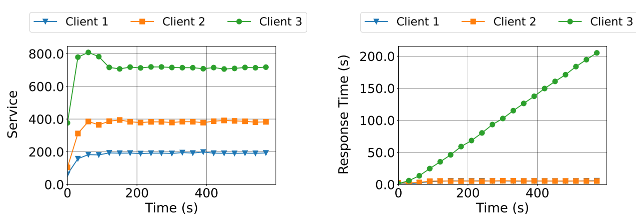
[7] There are in total 10000 tokens for KV cache that can be stored on GPU.



(a) Absolute difference for accumulated service

(b) Received service rate, calculated as an average of 60s time windows. (VTC)

Figure 3: Two clients with different request rates and both overloaded. Client 1 sends 90 requests per minute. Client 2 sends 180 requests per minute, both evenly spaced out so that each request is sent at a consistent time interval throughout the minute. Every request has input lengths of 256 and output lengths of 256. Both clients are backlogged because they exceed the server capacity.



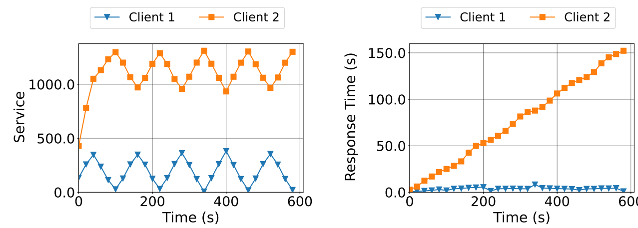
(a) Received service rate (VTC).

(b) Response time (VTC).

Figure 4: Client 3 who is overloaded can consume more than its share as Clients 1 and 2 are sending requests lower than their share. Clients 1, 2, and 3 send 15, 30, and 90 requests per minute, respectively, under uniform distribution. Requests have input lengths of 256 and output lengths of 256. Client 3 is backlogged, while Clients 1 and 2 are not.

reflecting the ideal case's fairness. We experiment from the simplest setting, where clients send requests following a uniform distribution with the same input and output length, to complex settings, where requests arrive stochastically, with various input and output lengths.

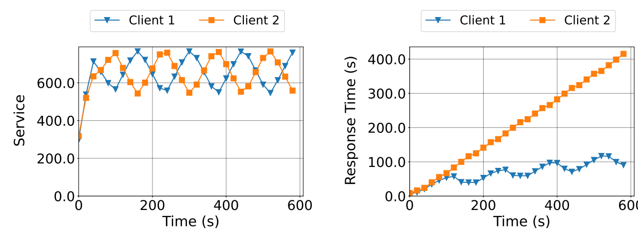
Constant request rates We start with scenarios where requests arrive deterministically with the same input and output length. In Figure 3, two clients send requests at different rates, but are both constantly overloaded. In this case, Figure 3a shows VTC can keep the difference between services received by both clients to be small. FCFS cannot maintain fairness, which always serves more for the client who is sending requests at a higher rate. Figure 3b shows the real-time received service rate for two clients in VTC, which confirms that the two received the same level of services at any time interval. This experiment empirically validates Theorem 4.4.



(a) Received service rate (VTC).

(b) Response time (VTC).

Figure 5: ON/OFF request pattern. Client 1 sends 30 requests per minute (less than half of the capacity) during the ON phase and switches to OFF phase periodically. Client 2 is always in the ON phase, sending requests at a rate of 120 requests per minute (larger than half of the capacity). Requests have input lengths of 256 and output lengths of 256.



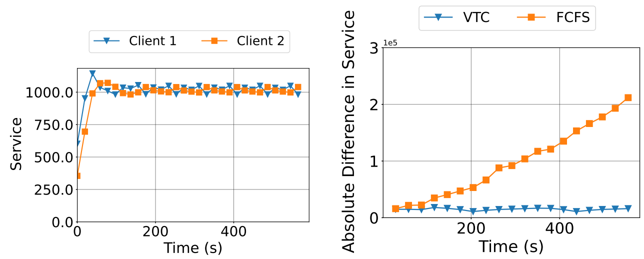
(a) Received service rate (VTC).

(b) Response time.

Figure 6: ON/OFF request pattern. Client 1 sends 120 requests per minute constantly during the ON phase (over its share), and stops sending during the OFF phase. Client 2 sends 180 requests per minute all the time (over its share). Requests have input lengths of 256 and output lengths of 256.

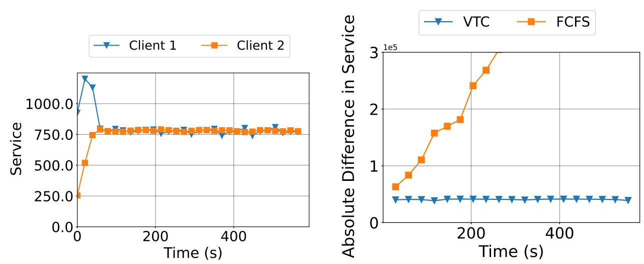
In Figure 4, three clients send requests at around $2/13$, $4/13$, and $> 7/13$ of the server's capacity, respectively. In this case, Clients 1 and 2 can be served immediately when their requests arrive (Figure 4b), and Client 3 will consume the remaining capacity (more than 1/3), which is an empirical illustration of the work-conserving property of VTC. The service received for Client 1 and Client 2 have a ratio $1:2$, which is consistent with their request rates (15 versus 30).

ON/OFF request pattern In real-world applications, clients usually do not always send requests to the server. They may occasionally be idle ("OFF" phase). We call this the "ON/OFF" pattern. In Figure 5, Client 2 is always in the "ON" phase, sending requests at a rate of 120 per minute. Client 1 sends 30 requests per minute (less than half of the capacity) during the ON phase and switches to OFF phase periodically. Since Client 1 uses less than half the system capacity when it is in the ON phase, its requests are mostly processed before it switches to the OFF phase (Figure 5b). When it is in the OFF phase, Client 1 thus takes all the system capacity. The total service rate remains the same, which confirms VTC's



(a) Received service rate (VTC). (b) Absolute difference for accumulated service.

Figure 7: Client 1 sends 480 requests per minute. Client 2 sends 90 requests per minute. Requests arrive according to a Poisson process with the coefficient of variance 1. Requests sent from Client 1 have input lengths of 64 and output lengths of 64. Requests sent from Client 2 have input lengths of 256 and output lengths of 256.



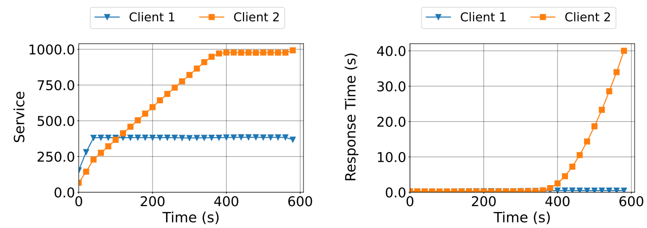
(a) Received service rate (VTC). (b) Absolute difference for accumulated service.

Figure 8: Client 1 sends 480 requests per minute. Client 2 sends 90 requests per minute. Requests arrive according to a Poisson process with the coefficient of variance 1. Requests sent from Client 1 have input lengths of 64 and output lengths of 512. Requests sent from Client 2 have input lengths of 512 and output lengths of 64.

flexibility in achieving work-conserving.

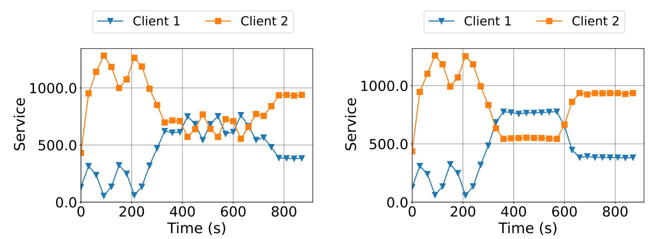
On the contrary, in Figure 6, client 1 sends much more than half the capacity during the ON phase, and makes itself always backlogged. Thus, even when it is in the OFF phase, it is still in the backlog status. In this case, Client 1 and Client 2 should still receive the same level of service rate.

Variable input/output length and poisson process In this experiment, we simulate scenarios where requests arrive stochastically. Furthermore, they send requests with different input and output lengths. In both Figure 7 and Figure 8, Client 1 sends requests with a high rate and Client 2 sends requests with a rate lower but still over its share. Requests arrive according to a Poisson process with the coefficient of variance 1. In Figure 7, client 1 sends short requests, and client 2 sends long requests. In Figure 8, Client 1 sends requests with short



(a) Received service rate (VTC). (b) Response time (VTC).

Figure 9: Client 1 sends 30 requests per minute, Client 2 sends 120 requests per minute, in a uniform arrival pattern. Requests have input lengths of 256 and output lengths of 256. Client 1 sends 30 requests per minute, which is under half of the server's capacity. Client 2 sends requests at a linearly increasing rate, and gradually over half of the system capacity.



(a) Received service rate (VTC). (b) Received service rate (LCF).

Figure 10: Clients send requests in three phases, all with uniform arrival patterns. The first 5 minutes is ON/OFF phase. Client 1 sends 30 requests per minute during the ON phase (less than its share) and stops sending during the OFF phase. Each ON or OFF phase has 60 seconds. The second 5 minutes is the overload phase. Both Client 1 and Client 2 send 60 requests per minute, which causes the server to be overloaded. In the last 5 minutes, Client 1 sends 30 requests per minute (less than its share), and Client 2 sends 90 requests per minute, which causes the server to be still overloaded. Requests all have input lengths of 256 and output lengths of 256.

input and long output, while Client 2 sends requests with long input and short output. Similarly, with the observation before, VTC maintains a bounded difference between the services received by two clients. FCFS cannot preserve fairness according to Figure 7b and Figure 8b. This confirms that VTC can work under stochastic workloads with variable lengths.

Isolation To illustrate the isolation property, we use the setup with a deterministic arrival pattern and the same input length and output length of 256. In Figure 9, Client 1 sends 30 requests per minute, which is under half of the server's capacity. Client 2 acts as an "ill-behaved" client. It sends requests at a linearly increasing rate, and gradually over half of the system capacity. We observe that the response time of requests from client 1 is roughly unchanged, empirically

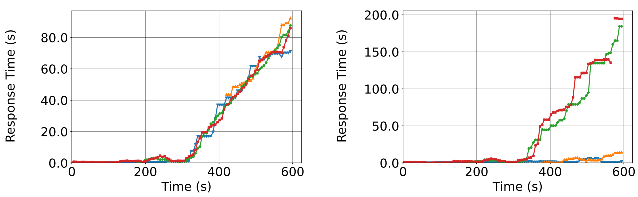


Figure 11: Response time of FCFS (Left) and VTC (Right) in real traces. Each curve corresponds to one client.

validating the property stated in Theorem 4.13.

Distribution shift In reality, clients' behavior may change over time. To this end, we evaluate the robustness of VTC when the distribution of client requests shifts. In Figure 10, we construct a 15-minute workload comprising three phases. The first phase is an ON/OFF phase, in which Client 1 sends requests less than its share only during the ON phase and stops during the OFF phase. Client 2 sends requests at a constant rate, which makes the server overloaded. We can observe the pattern for the first phase to be similar to Figure 5a, which maintains a constant total service. During the second phase, because the two clients both send requests over their share, a fair server should let them receive the same level of service. Figure 10a demonstrates that VTC yields a desired pattern, similar to that shown in Figure 3b. Figure 10b reveals that LCF disproportionately serves Client 1, as it inherits Client 1's deficit from the first phase. In the last phase, the serving pattern for VTC and LCF are similar, because they simply serve all requests from Client 1 immediately as Client 1 sends requests under its share.

5.3 Results on Real Workloads

We construct real workload traces from the traces of LMSYS Chatbot Arena [48, 49], following a similar process in [39]. The trace is from a server that serves multiple LLMs. To adapt it to our setting, we treat each LLM as a client. In total, there are 27 clients. To sample from this log, we define D, the duration, and R, the request rate. We then sample $R * D$ requests from the trace, and re-scale the real-time stamps to $[0, D]$. We use a duration of 10 minutes to be consistent with previous experiments, and a request rate of 150 requests per minute for the whole system. With the adapted workload, we run Llama-2-7b on A10G (24GB). For better visualization, we select two clients that send the most requests and two clients that send a medium number of requests. We sort 27 clients according to the number of requests they send, and depict the statistics of the $13^{th}, 14^{th}$ and $26^{th}, 27^{th}$ clients. We do not choose clients that send the least requests because they typically only send requests in a small interval.

Effect on response time Figure 11 shows the response time of different clients on the real trace. With FCFS scheduling, the response time of all clients increases drastically because some clients send over their share, monopolizing the service and impacting other clients. With VTC, only clients that send requests over its share will have a drastic increase in the response time.

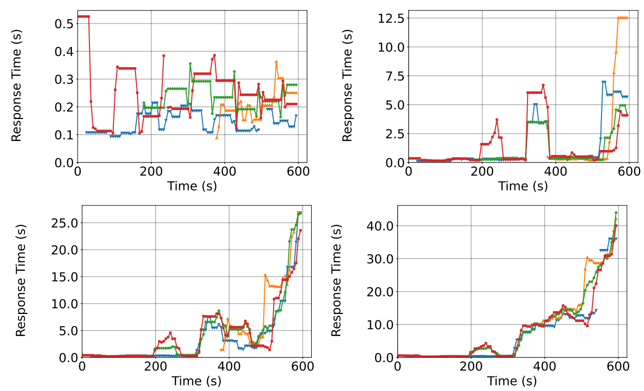


Figure 12: Response time of RPM in real traces. Left to right corresponds to a different rate limit (5, 10, 15, 20 requests per minutes, respectively).

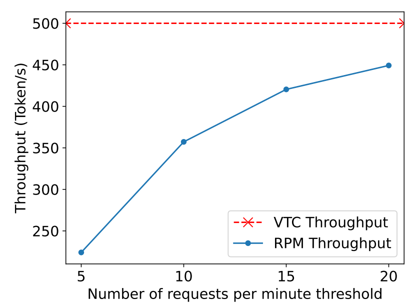
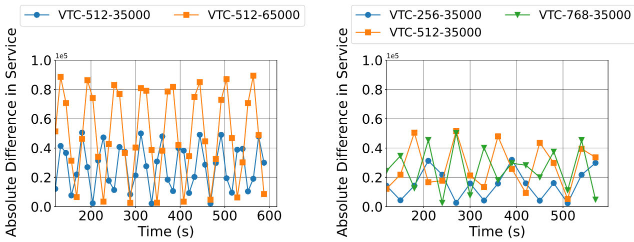


Figure 13: Throughput of RPM versus different number of requests per minute threshold. Compared with VTC, RPM consistently exhibits a lower throughput.

Analysis of request rate per limit approach In Figure 12, we show the response of RPM approach with different rate limits. In Figure 13, we show the corresponding throughput comparison with VTC. These plots reveal a core dilemma of the RPM approach - the system has to choose between fairness or throughput, but not both. If the rate limit is low, then the system rejects many requests from clients that send over their share. This opens the capacity for clients with fewer requests. As demonstrated in the uppermost plot, all requests have a similar response time. However, this low rate limit rejects more requests than needed, causing a lower throughput (cluster-wise throughput is ≈ 220 output tokens per second when RPM=5, as opposed to ≈ 500 tokens per second in VTC or FCFS). When the rate limit is set higher, the system throughput is gradually increasing, i.e., increasing from 224 tokens to 450 tokens per second. However, the response time for all requests grows up. When the request rate is set higher



(a) Different memory pool size.

(b) Different request length.

Figure 14: In all settings, both clients are sending requests of the same lengths with uniform arrival patterns. They send requests with different request rates but are both backlogged. Three different request lengths (256*2, 512*2 and 768*2) are evaluated for the 35000 KV cache setting.

and higher, the response time curve converges to the one in FCFS, and there is no fairness guarantee anymore. In other words, the RPM approach can be summarized as follows: it functions as an FCFS (First-Come, First-Served) approach with admission control (rate limiting), rather than as a truly fair scheduler. Its fairness is achieved by rejecting numerous requests from other clients, which compromises the overall system throughput.

5.4 Ablation Study

In Figure 14, we evaluate how different memory pool sizes and request lengths will affect scheduling fairness. As shown in Figure 14a, with a larger memory pool size, the attainable batch size becomes larger. Therefore, there is greater variation in the absolute difference of accumulated services received by the clients when the memory pool is 65000 than that is 35000, which empirically validates Theorem 4.4. Figure 14b demonstrates that larger request lengths will also lead to greater variations in the service difference. This is caused by the unknown output length of request generation. At line 19 in Algorithm 2, the most conservative way of only counting the input tokens leads to over-compensation for the smallest counter, as all the potential output tokens are not counted. A shorter request length has a milder effect of over-compensation. The curves of $(512*2)$ and $(768*2)$ show the same variance. This is because at length $(512*2)$, the upper bound given by VTC has been reached.

6 Related Works

Fairness in scheduling Achieving fairness in scheduling resources in a multi-client environment has been a long-standing topic in computer science [11, 37, 38, 47]. Among these, Fair Queuing [26] has been adapted into many variants for different contexts such as CPU scheduling [2], link bandwidth allocation [8, 12, 14, 19, 34], and memory allocation [29]. Deficit round robin [41] and stochastic fair queuing [23] are non-real-time fair queuing algorithms for variable-size packets, providing guarantees for long-term fairness. There are also real-time fair queuing algorithms (e.g., WFQ [8] and SFQ [14]) that can make more strict short-term delay guarantees [9]. Our scheduling algorithm is different from these algorithms because we need to consider the batching effects across multiple clients' requests and deal with unknown request length. Further, we need to accommodate a flexible notion of fairness on both performance and GPU resource consumption.

Fairness in ML training Within the realm of deep learning, research has delved into scheduling jobs in shared clusters [5, 22, 28, 36], with a primary focus on long-duration training jobs. Machine Learning training jobs have unique characteristics and traditional fair schedulers [15, 18] designed for big-data workflow usually fail [22]. In particular, Themsis [22] points out that ML jobs are device placement sensitive, where jobs will be envious of other's placement even if they are assigned the same number of resources. It then defines a finish-time fairness metric to measure fairness in ML training scenarios. Pollux [36] further points out that ML jobs should jointly consider the throughput and the statistical efficiency, and develop a goodput-based scheduler that further improves the finish-time fairness of ML jobs. In this paper, we consider fairness in LLM serving. The fairness problem in LLM serving is quite different from the fairness problem in model training. In model training, different clients' GPUs are isolated and the problem is which GPUs are assigned to each client. Achieving fairness in LLM serving requires design for a different set of issues, including how to batch requests from multiple clients to achieve high GPU utilization.

LLM Serving Systems How to improve the performance of LLM serving systems has recently gained significant attention. Notable techniques cover advanced batching mechanisms [10, 46], memory optimizations [20, 40], GPU kernel optimizations [1, 6, 30, 44], model parallelism [1, 21, 35], parameter sharing [50], and speculative execution [24, 42] were proposed. FastServe [45] explored preemptive scheduling to minimize job completion time (JCT). However, none of these works consider fairness among clients. Our work bridges this gap, and our proposed scheduling methods can be easily integrated with many of these techniques. Our implementation has already integrated with continuous batching (iteration-level scheduling) [16, 46] and PagedAttention [20].

7 Conclusion

We studied the problem of fair serving in Large Language Models (LLMs) with regard to the service received by each client. We identified unique characteristics and challenges associated with fairness in LLM serving, as compared to traditional fairness problems in networking and operating systems. We then defined what constitutes fairness and proposed a fair scheduler, applying the concept of fair sharing to the domain of LLM serving at the token granularity.

8 Acknowledgment

We thank Aurick Qiao, Shu Liu, Stephanie Wang, Hao Zhang for helpful discussions and feedback. This research was supported by gifts from Anyscale, Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, and VMware. Ying is partly supported by the Stanford Center for Automated Reasoning. We thank Clark Barrett for academic advising and funding support.

References

[1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[2] Jens Axboe. Linux block io—present and future. In *Ottawa Linux Symp*, pages 51–61, 2004.

[3] Dimitri Bertsekas and Robert Gallager. *Data networks*. Athena Scientific, 2021.

[4] Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoqiang Zheng. Group ratio round-robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems. In *USENIX Annual Technical Conference (ATC)*, pages 337–352. USENIX, 2005.

[5] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[6] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[7] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12, 1989.

[8] Alan J. Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In Lawrence H. Landweber, editor, *ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, pages 1–12. ACM, 1989.

[9] Peter L Dorlan. *An introduction to computer networks*. Autoedición, 2016.

[10] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[11] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of Networks and Systems Design and Implementation (NSDI)*, 2011.

[12] S. Jamaloddin Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings IEEE INFOCOM '94, The Conference on Computer Communications, Thirteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Networking for Global Communications*, pages 636–646. IEEE Computer Society, 1994.

[13] P. Goyal, H.M. Vin, and Haichen Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, 1997.

[14] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 157–168. ACM, 1996.

[15] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in Multi-Resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, Savannah, GA, November 2016. USENIX Association.

[16] Hugging Face. Text generation inference. https://github.com/huggingface/text-generation-inference. Accessed: 2023-11.

[17] HuggingFace. Text-generation-inference(tgi). https://github.com/huggingface/text-generation-inference, 2023.

[18] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 261–276. Association for Computing Machinery, 2009.

[19] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 37–48. ACM, 2004.

[20] 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*, pages 611–626, 2023.

[21] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[22] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient gpu cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[23] P.E. McKenney. Stochastic fairness queueing. In *Proceedings. IEEE INFOCOM '90: Ninth Annual Joint Conference of the IEEE Computer and Communications Societies@m_The Multiple Facets of Integration*, pages 733–740 vol.2, 1990.

[24] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 2023.

[25] ModelTC. Lightllm: Python-based llm inference and serving framework. https://github.com/ModelTC/lightllm, 2023. GitHub repository.

[26] J. Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, 35(4):435–438, 1987.

[27] Deepak Narayanan, Keshav Santhanam, Peter Henderson, Rishi Bommasani, Tony Lee, and Percy Liang. Cheaply estimating inference efficiency metrics for autoregressive transformer models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[28] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[29] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 208–222, 2006.

[30] NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer.

[31] OpenAI. Openai api reference. https://platform.openai.com/docs/api-reference. Accessed: 2023-11.

[32] OpenAI. Gpt-4 turbo, 2023.

[33] OpenAI. Rate limit. https://platform.openai.com/docs/guides/rate-limits?context=tier-free, 2023.

[34] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[35] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

[36] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

[37] Bozidar Radunovic and Jean-Yves Le Boudec. A unified framework for max-min and min-max fairness with applications. *IEEE/ACM Transactions on networking*, 15(5):1073–1083, 2007.

[38] Uwe Schwiegelshohn and Ramin Yahyapour. Fairness in parallel job scheduling. *Journal of Scheduling*, 3(5):297–320, 2000.

[39] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

[40] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: high-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[41] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 4(3):375–385, 1996.

[42] Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. Blockwise parallel decoding for deep autoregressive models. *Advances in Neural Information Processing Systems*, 31, 2018.

[43] Ion Stoica, Hussein M. Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 288–299. IEEE Computer Society, 1996.

[44] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. Lightseq: A high performance inference library for transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*, pages 113–120, 2021.

[45] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[46] 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.

[47] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.

[48] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric Xing, et al. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *arXiv preprint arXiv:2309.11998*, 2023.

[49] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Conference on Neural Information Processing Systems, Datasets and Benchmarks Track*, 2023.

[50] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. Pets: A unified framework for parameter-efficient transformers serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 489–504, 2022.

A Missing Proofs in Proving Fairness of VTC

Lemma A.1. *In Algorithm 2, $\min_{i\in Q}(c_i)$ is non-decreasing during the time when $Q \neq \emptyset$.*

Proof. We prove the lemma by case study on each line of changing the c_i's.

- In the initialization, all $c_i = 0$, lemma holds.
- If the condition of line 7 is satisfied, at lines 8-14, a new client will be added to Q. If lines 10-11 are reached, the $\min_{i\in Q}(c_i)$ is equals to its value at the last time when $Q \neq \emptyset$. If line 13 is reached, since $c_u = \max\{c_u, \min_{i\in Q} c_i\}$, the $\min_{i\in Q}(c_i)$ will not change.
- At line 23 and line 29, the c_i's can only increase, so that $\min_{i\in Q}(c_i)$ is non-decreasing.
- At line 25, if a client has cleared all its requests from Q, that the client is removed from Q, $\min_{i\in Q}(c_i)$ cannot decrease.

□

Lemma A.2. *The following invariant holds at any time in Algorithm 2 when $Q \neq \emptyset$:*

$$\max_{i\in Q} c_i - \min_{i\in Q} c_i \leq \max(w_p \cdot L_{input}, w_q \cdot M) \tag{2}$$

Proof. We prove the lemma by induction. During the induction, for each line of change of c_i in Algorithm 2, we use c_i' to denote the new value and c_i to denote the original value. Similarly, we use Q' to donate the new value and Q to denote the original value. We also use $c_i^{(t)}$ to denote the value of c_i at time t, and $Q^{(t)}$ to denote the value of Q at time t.

1. In the initialization, all $c_i = 0$, Equation (2) holds.

2. If a client $u \notin Q$ receive a new request and thus $Q' = Q \cup \{u\}$, line 8-9 will be reached, and thus $c_u' = \max\{c_u, \min_{i\in Q} c_i\} \geq \min_{i\in Q} c_i$. Then we have,

$$\min_{i\in Q'} c_i' = \min\{c_u', \min_{i\in Q} c_i\} = \min_{i\in Q} c_i. \tag{4}$$

   Let t be the last time that u was in Q before the change, and thus $c_u^{(t)} = c_u$. From Equation (2), there is

$$\max_{i\in Q^{(t)}} c_i^{(t)} - \min_{i\in Q^{(t)}} c_i^{(t)} \leq \max(w_p \cdot L_{input}, w_q \cdot M).$$

   Then we have

$$c_u^{(t)} \leq \max_{i\in Q^{(t)}} c_i^{(t)} \leq \min_{i\in Q^{(t)}} c_i^{(t)} + \max(w_p \cdot L_{input}, w_q \cdot M).$$

   From Theorem A.1, there is $\min_{i\in Q^{(t)}} c_i^{(t)} \leq \min_{i\in Q} c_i$, so we have

$$c_u = c_u^{(t)} \leq \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M),$$

   which can derive

$$c_u' = \max\{c_u, \min_{i\in Q} c_i\} \leq \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M).$$

   Combine with Equation (2) and Equation (4), there is

$$\begin{aligned}\max_{i\in Q'} c_i' &= \max\{c_u', \max_{i\in Q} c_i\} \\ &\leq \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M) \\ &\leq \min_{i\in Q'} c_i' + \max(w_p \cdot L_{input}, w_q \cdot M)\end{aligned}$$

   Therefore, Equation (2) holds after the change.

3. If a client u is left from Q at line 21, the difference $\max_{i\in Q} c_i - \min_{i\in Q} c_i$ will not increase. Because $\max(C') - \min(C') \leq \max(C) - \min(C), \forall C \supseteq C', C' \neq \emptyset$. Therefore, Equation (2) still holds.

4. At line 19, since $c_k = \min_{i\in Q} c_i$, there is

$$\min_{i\in Q} c_i \leq \min_{i\in Q} c_i' \leq c_k' \leq \min_{i\in Q} c_i + w_p \cdot L_{input}. \tag{5}$$

   From Equation (2), we have

$$\max_{i\in Q} c_i \leq \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M).$$

   Because:

$$\max_{i\in Q} c_i' = \max(\max_{i\in Q} c_i, c_k')$$

   We have:

$$\max_{i\in Q} c_i' \leq \max(\min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M), c_k') \tag{6}$$

   In Equation (5) we have derived that:

$$c_k' \leq \min_{i\in Q} c_i + w_p \cdot L_{input}$$

   Thus:

$$\begin{aligned}c_k' &\leq \min_{i\in Q} c_i + w_p \cdot L_{input} \\ &\leq \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M)\end{aligned}$$

   Thus:

$$\begin{aligned}&\max(\min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M), c_k') = \\ &\min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M).\end{aligned}$$

   Thus Equation (6) gives:

$$\max_{i\in Q} c_i' \leq \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M) \tag{7}$$

   Finally, combining the inequality from Equation (5) that

$$\min_{i\in Q} c_i \leq \min_{i\in Q} c_i',$$

   we arrive at:

$$\max_{i\in Q} c_i' \leq \min_{i\in Q} c_i' + \max(w_p \cdot L_{input}, w_q \cdot M).$$

   Therefore, Equation (2) holds.

5. At line 25, let $k = \arg\max_{i\in Q} c'_i$, so that $c'_k = \max_{i\in Q} c'_i$. Let r be the last one among requests from k that have been scheduled. Let t be the time when r was selected at line 16. Since r is the last one been scheduled from k, there is
$$\max_{i\in Q} c'_i = c'_k \le c_k^{(t)} + w_q \cdot M \tag{8}$$
Because request r from client k has been scheduled at time t, from line 16, there is $c_k^{(t)} = \min_{i\in Q^{(t)}} c_i^{(t)}$. From Lemma A.1, we have $\min_{i\in Q^{(t)}} c_i^{(t)} \le \min_{i\in Q} c'_i$. Combine with Equation (8), we have
$$\max_{i\in Q} c'_i - \min_{i\in Q} c'_i \le w_q \cdot M.$$
Therefore, Equation (2) holds.

□

Theorem 4.8. *For any work-conserving schedule without preemption, there exists some query arrival sequence such that for client f, g and a time period t_1, t_2, such that*
$$|W_f(t_1,t_2) - W_g(t_1,t_2)| \ge w_q \cdot M,$$
where clients f, g are backlogged during the time $[t_1,t_2)$.

Proof. Consider at time 0 the client f sends a list of requests which cannot fit in the memory at once. Because of work-conserving, client f will fill the whole running batch. In this case, client f is backlogged, and any new query is not processed until the existing queries finish processing. Assume that all existing queries finish at time T, and that at time ε with ε close to 0, a second client g sends another batch of requests. Now during the time interval $[\varepsilon, T]$, both clients f, g are backlogged since there exist queries from both clients in the queue. At time T, client f received service from the first batch of processing, which can be up to $w_q \cdot M$ if the memory is luckily fully utilized. Thus we have
$$W_f(\varepsilon, T) = w_q \cdot M.$$
On the other hand, client g did not receive any service during the time period $[\varepsilon, T]$. Thus $W_g(\varepsilon, T) = 0$. In this case, we have constructed an instance with
$$|W_f(t_1,t_2) - W_g(t_1,t_2)| \ge w_q \cdot M.$$

□

Theorem 4.11. *Let $A(r)$ and $D(r)$ denote the arrival time and dispatch time of a request r. Assume there are in total n clients, $\forall t_1, t_2$, if at t_1, a client f is not backlogged and has no requests in the running batch, then the next request r_f with $t_1 < A(r_f) < t_2$ will have its response time bounded:*
$$D(r_f) - A(r_f) \le 2 \cdot (n-1) \cdot \frac{\max(w_p \cdot L_{input}, w_q \cdot M)}{a} \tag{3}$$
Here a is the lower bound of the capacity in Definition 4.10.

Proof. Let the counter for f be c_f after line 9 for r_f. Before D_{r_f}, since r_f is always in the queue, the counter for f will not be lifted. Since there is no running batch of f in the server, line 17 will select r_f to be the next one for f. Lemma 4.3 shows that for any other client g,
$$c_g - c_f < \max(w_p \cdot L_{input}, w_q \cdot M).$$
In the worst case where these counters are incremented sequentially, it will take at most $2*(n-1)*\frac{\max(w_p \cdot L_{input}, w_q \cdot M)}{a}$. Thus, giving a bound for the dispatch time of r_f. □

Theorem 4.9. *If a client f is backlogged during time interval $[t_1,t_2)$, for any client g, there is*
$$W_f(t_1,t_2) \ge W_g(t_1,t_2) - 4U.$$
Here U is the upper bound from Equation (2).

Proof. If g is not backlogged during the entire $[t_1,t_2)$, then $W_g(t_1,t_2) \le U$, the theorem trivially holds. Next, assume g is backlogged at some point during $[t_1,t_2)$. Let t'_1, t'_2 be the first time and the last time g is backlogged between $[t_1,t_2)$. Since there is no request submitted in $[t_1,t'_1)$ and $[t'_2,t_2)$, we have
$$W_g(t_1,t'_1) \le U, \quad W_g(t'_2,t_2) \le U. \tag{9}$$
Since c_i's in Algorithm 2 are non-decreasing,
$$c_f^{(t_1)} \le c_f^{(t'_1)} \le c_f^{(t'_2)} \le c_f^{(t_2)}, \tag{10}$$
$$c_g^{(t_1)} \le c_g^{(t'_1)} \le c_g^{(t'_2)} \le c_g^{(t_2)}. \tag{11}$$
According to Lemma 4.3 there is,
$$c_g^{(t'_2)} \le c_f^{(t'_2)} + U, \quad c_g^{(t'_1)} \ge c_f^{(t'_1)} - U.$$
By Equation (10) and Equation (11):
$$c_g^{(t'_2)} \le c_f^{(t_2)} + U, \quad c_g^{(t'_1)} \ge c_f^{(t_1)} - U.$$
Since $W_g(t'_1,t'_2) \le c_g^{(t'_2)} - c_g^{(t'_1)}$, there is
$$W_g(t'_1,t'_2) \le c_f^{(t_2)} - c_f^{(t_1)} + 2U.$$
Combine with Equation (9), there is:
$$\begin{aligned} W_g(t_1,t_2) &= W_g(t_1,t'_1) + W_g(t'_1,t'_2) + W_g(t'_2,t_2) \\ &\le c_f^{(t_2)} - c_f^{(t_1)} + 4U. \end{aligned}$$
Since f is backlogged during (t_1,t_2),
$$W_f(t_1,t_2) = c_f^{(t_2)} - c_f^{(t_1)}$$
Thus:
$$W_f(t_1,t_2) \ge W_g(t_1,t_2) - 4U$$

□

Theorem 4.13. *(Fairness for non-overloaded clients) For any time interval* $[t_1, t_2)$, *we claim the following.*

Assume a client f *is not backlogged at time* t_1 *and for any time interval* $[t, t_2), t_1 \leq t < t_2$, f *has requested services less than* $\frac{T(t,t_2)}{n(t,t_2)} - 5U$, *where* $T(t, t_2)$ *is the total services received for all clients during the interval* $[t, t_2)$, $n(t, t_2)$ *is the number of clients that have requested services during the interval, and* U *is the upper bound from Equation* (2).

Then, all of the services requested from f *during the interval* $[t_1, t_2)$ *will be dispatched.*

Proof. We prove by contradiction. Assume there is a request from f that has not been dispatched in t_2, i.e., f is backlogged at t_2. Since f is not backlogged at t_1, there exists a (non-empty) set of time steps such that f becomes backlogged. We let t be the largest element in the set, i.e. f is backlogged at any time in $[t, t_2)$. We claim that $W_f(t, t_2) \geq \frac{T(t,t_2)}{n(t,t_2)} - 4U$.

From the pigeonhole principle, there is at least one client g who has received services $W_g(t, t_2) \geq \frac{T(t,t_2)}{n(t,t_2)}$. If $f = g$, the claim holds. If not, from Theorem 4.9, we have

$$W_f(t, t_2) \geq W_g(t, t_2) - 4U \geq \frac{T(t, t_2)}{n(t, t_2)} - 4U.$$

Since f swicthes from non-backlogged to backlogged at t, requests sent before t at most contributes a U increase in $W_f(t, t_2)$. Thus, requests sent in (t, t_2) at least contribute to $\frac{T(t,t_2)}{n} - 5U$, which contradicts to the assumption in the theorem. □