Algorithm 1 KVCache-centric Scheduling Algorithm

Input: prefill instance pool P, decoding instance pool D, request R, cache block size B.
Output: the prefill and decoding instances (p, d) to process R.

$block\_keys \leftarrow$ PrefixHash$(R.prompt_tokens, B)$
$TTFT \leftarrow \inf$
$p \leftarrow \emptyset$
$best\_prefix\_len, best\_matched\_instance \leftarrow$ FindBestPrefixMatch$(P, block_keys)$
for $instance \in P$ **do**
    $prefix\_len \leftarrow instance.prefix\_len$
    $T_{queue} \leftarrow$ EstimatePrefillQueueTime$(instance)$
    if $\frac{best\_prefix\_len}{prefix\_len} <$ **kvcache_balancing_threshold then** ▷ Cache-aware prefill scheduling
        $T_{prefill} \leftarrow$ EstimatePrefillExecutionTime(len$(R.prompt_tokens), prefix_len)$
        if $TTFT > T_{queue} + T_{prefill}$ **then**
            $TTFT \leftarrow T_{queue} + T_{prefill}$
            $p \leftarrow instance$
        end if
    else ▷ Cache-aware and -balancing prefill scheduling
        $transfer\_len \leftarrow best\_prefix\_len - prefix\_len$
        $T_{transfer} \leftarrow$ EstimateKVCacheTransferTime$(instance, best_matched_instance, transfer_len)$
        $T_{prefill} \leftarrow$ EstimatePrefillExecutionTime(len$(R.prompt_tokens), best_prefix_len)$
        if $TTFT > T_{transfer} + T_{queue} + T_{prefill}$ **then**
            $TTFT \leftarrow T_{transfer} + T_{queue} + T_{prefill}$
            $p \leftarrow instance$
        end if
    end if
end for
$d, TBT \leftarrow$ SelectDecodingInstance(D) ▷ Load-balancing decoding scheduling
if $TTFT > TTFT\_SLO$ **or** $TBT > TBT\_SLO$ **then**
    reject R; **return**
end if
if $\frac{best\_prefix\_len}{p.prefix\_len} >$ **kvcache_balancing_threshold then**
    TransferKVCache$(best_matched_instance, p)$ ▷ KVCache hot-spot migration
end if
return (p, d)

6 KVCache-centric Scheduling

In this section, we mainly discuss how Conductor schedules the requests and KVCache blocks under normal conditions, leaving the discussion on overload scenarios for the next section.

6.1 Prefill Global Scheduling

Previous research on LLM serving typically uses a load-balancing strategy that evaluates the load on each instance based on the number of assigned requests. In Mooncake, however, the selection of prefill instances considers additional factors—not just load but also the prefix cache hit length and the distribution of reusable KVCache blocks. While there is a preference to route requests to prefill instances with longer prefix cache lengths to reduce computation costs, it may be beneficial to schedule them to other nodes to ensure overall system balance and meet TTFT SLOs. To address these complexities, we propose a cache-aware global scheduling algorithm that accounts for both the prefill time due to the prefix cache and the queuing time associated with the load on the instance.

Algorithm 1 details the mechanism for our cache-aware prefill scheduling. For every new request, its input tokens are divided into several blocks, and a hash key is computed for each block. This involves generating a hash key of tokens in a block concatenated with the hash key of the previous block (if available). The request's block keys are then compared one by one against each prefill instance's cache keys to identify the prefix match length ($prefix\_len$). Similar reuse logic is already implemented in vLLM, but the open-source version of vLLM only supports local KVCache caching.

With this matching information, Conductor estimates the corresponding execution time based on the request length and $prefix\_len$ (which varies by instance). It then adds the estimated waiting time for that request to get the TTFT on that instance. Finally, Conductor assigns the request to the instance