AFFIX CACHE FOR DIFFUSION LARGE LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion Large Language Models (DLLMs) enable non-autoregressive decoding, but efficient inference support remains immature: unlike autoregressive models, whose requests reuse a shared prefix key-value (KV) cache, DLLMs use bidirectional attention, so a shared context's KV states depend on the tokens still being decoded, leaving directly reused caches stale and full recomputation necessary. We present **ACache**, a cross-request cache reuse mechanism for shared spans, or *affixes*, at any position: prefix, infix, or suffix. ACache measures the influence of affix tokens on the masked generation region to identify a small request-specific subset as *Anchor Tokens*, and recomputes only their KV states while reusing the remaining affix cache. Built on state-of-the-art intra-request caching mechanisms, ACache recovers most of the accuracy lost to direct affix-cache reuse on average when recomputing around 20% of affix tokens, and at that budget preserves more accuracy than selection criteria adapted from prior cross-request cache-reuse systems. We co-design ACache with a modern inference engine, whose attention reads each request's recomputed Anchor KV states alongside one affix cache shared across concurrent requests. Against the same system with only intra-request caching, ACache cuts recompute latency by up to 56.7%, translating to as much as $1.71\times$ end-to-end throughput, while reducing peak KV cache memory by up to 45.8%.

1 INTRODUCTION

Large Language Models (LLMs) have traditionally been autoregressive (AR); Diffusion Large Language Models (DLLMs) (Nie et al., 2025b; Ye et al., 2025a) instead decode by iterative denoising, predicting masked positions in arbitrary order over a context that attention spans bidirectionally, which enables low-latency parallel decoding (Israel et al., 2025; Ben-Hamu et al., 2025; Wu et al., 2026b) and gains in data efficiency and reasoning (Ni et al., 2025; He et al., 2026a).

For AR LLMs (Figure 1, left), prefix caching is a well-established mechanism for efficiency: requests reuse a key-value (KV) cache for shared prefixes such as system prompts and prompt templates (Gim et al., 2024; Zheng et al., 2024). In contrast, DLLMs use bidirectional attention, so the KV states of a shared prefix depend on the tokens still being decoded, and a directly reused prefix cache goes stale (Figure 1, middle). As a result, practical DLLM inference frameworks (Ma et al., 2025a; Liu et al., 2026c; Wu et al., 2026b; Jiang et al., 2026; Nguyen-Tri et al., 2026; Xiong et al., 2026) confine reuse to a single request (intra-request reuse), recomputing KV states at intervals to maintain context consistency; reusing caches directly across requests substantially degrades quality (Section 5).

To address this limitation, we propose ACache (Affix Cache), a cross-request cache reuse mechanism for DLLMs. We study shared contiguous text spans, which we call *affixes*, as a span may be a prefix, infix, or suffix of the context. Unlike text cached by AR LLMs, which always precedes generation, a suffix follows the generation region, as in infilling-style prompts (Lou et al., 2024; Xu et al., 2026). We observe that the influence of an affix's tokens on the masked generation region concentrates on a small subset, and that this subset stays stable as decoding proceeds. ACache exploits this finding (Figure 1, right): it reuses the shared affix cache while recomputing only that subset, which we term *Anchor Tokens*. ACache selects them once per request by mask-to-affix attention importance.

ACache composes with intra-request caching mechanisms rather than replacing them, and brings to DLLMs the cross-request cache reuse on which many AR inference-stack optimizations build (Zheng et al., 2024; Srivatsa et al., 2025; Pan et al., 2025; Qin et al., 2025; Yao et al., 2025; Yuan et al., 2026;

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

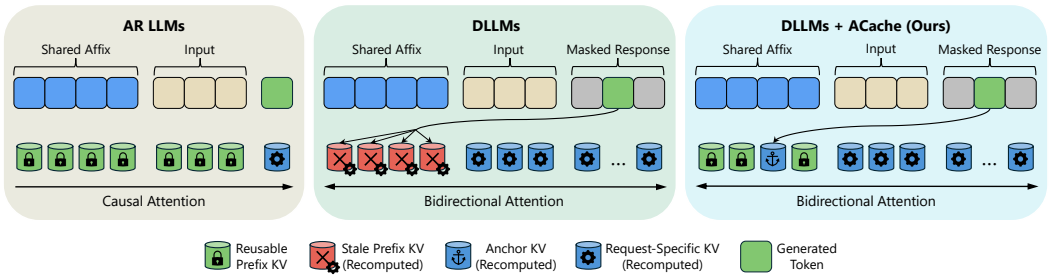


Figure 1: **Cache reuse in AR LLMs and DLLMs.** *Left*: AR LLMs reuse a shared prefix cache directly. *Middle*: in DLLMs, bidirectional attention makes that cache depend on the tokens still being decoded, leaving it stale. *Right*: ACache recomputes only the Anchor Tokens' KV states.

Ouyang et al., 2026; Wang et al., 2026b). We build ACache on Nano-vLLM, a vLLM-style engine with paged memory management (Kwon et al., 2023; Nano-vLLM, 2025), and co-design the engine with the algorithm: since within the affix only the Anchor KV differs across requests, the engine stores one affix cache, and a per-request read-slot map lets attention read it alongside the request's own Anchor KV. We also extend the engine to reuse that cache at different offsets and to interleave recomputation with decoding at step granularity (Yu et al., 2022; vLLM-Omni, 2026).

To summarize, our key contributions are as follows:

(i) We present ACache, a cross-request cache reuse mechanism for shared spans in DLLMs beyond prefixes, recomputing only the Anchor Tokens it selects by mask-to-affix importance.

(ii) We implement ACache on top of two state-of-the-art intra-request DLLM caching frameworks, dKV-Cache (Ma et al., 2025a) and Fast-dLLM (Wu et al., 2026b), where recomputing around 20% of affix tokens recovers most of the accuracy lost to direct reuse on average.

(iii) We co-design an engine that shares one affix cache: up to 56.7% lower recompute latency, $1.71\times$ end-to-end throughput, and 45.8% lower peak KV cache memory over intra-request caching.

2 Background & Motivation

Diffusion large language models. DLLMs now scale to billions of parameters (Nie et al., 2025a; Inception Labs et al., 2025; Song et al., 2025). LLaDA (Nie et al., 2025b) trains a diffusion model from scratch, while Dream (Ye et al., 2025a), DiffuLLaMA (Gong et al., 2025), and TESS 2 (Tae et al., 2025) start from pretrained AR LLMs; both routes remain competitive on general, mathematical, and coding tasks, and keep improving (Zhu et al., 2025; 2026a; Nie et al., 2026). A separate line of models instead adopts block diffusion (Han et al., 2023; Arriola et al., 2025), modeling tokens bidirectionally within a block and causally across blocks (Bie et al., 2025; Wu et al., 2026a; Fu et al., 2026; Cheng et al., 2026; DiffusionGemma Team et al., 2026). That structure keeps AR-style prefix caching workable, but confines both bidirectional context and parallelism to a block. ACache targets standard DLLMs, whose bidirectional attention spans the full sequence.

Caching techniques in DLLMs. DLLMs update a masked generation region over many decoding steps (Nie et al., 2025b; Ye et al., 2025a), paying for full-sequence attention per step, motivating cross-step KV reuse. Unlike AR LLMs, DLLMs are unable to maintain causal decoding's append-only KV cache (Pope et al., 2023): with bidirectional attention, KV states change during generation, prompt tokens included, so entries go stale rather than accumulate (Nguyen-Tri et al., 2026). Recent work therefore explores approximate caching: one line reuses KV states or intermediate features across steps on a predetermined schedule, varying the recomputation amount (Ma et al., 2025b; Liu et al., 2026c; Hu et al., 2026). dKV-Cache (Ma et al., 2025a) carries over already-decoded tokens' KV states with a one-step delay, while Fast-dLLM (Wu et al., 2026b) retains KV states within a decoding block (Table 1). A second line replaces the predetermined schedule with a runtime signal, selecting recomputation or eviction timing and layer or token granularity (Huang et al., 2026; Song et al., 2026; Cheong et al., 2026; Nguyen-Tri et al., 2026; Jiang et al., 2026; Wu et al., 2026d; Frumkin et al., 2026; Yang et al., 2026; Xiong et al., 2026; Lee et al., 2026a; He et al., 2026b). Both lines remain intra-request. Concurrently, BiCache (Go et al., 2026) reuses shared prefix KV states across requests in shallow layers where they remain stable, whereas ACache selects tokens rather than layers.

Table 1: **Cache recomputation.** Full-sequence attention frequency and cross-request prefix cache reuse.

Method	Full-sequence attention	Cross-req. cache reused?
AR LLM	**Once**, at prefill	Yes
dKV-Cache	**Every few steps**	No
Fast-dLLM	**Every block** boundary	No

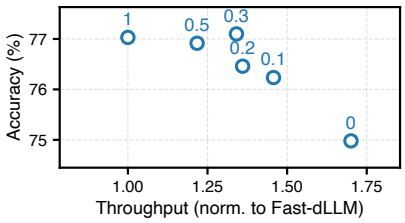


Figure 2: **Quality–throughput tradeoff.** ACache's Fast-dLLM-based system on LLaDA/GSM8K, 2-shot prefix, batch 16. Labels: Recompute ratios.

Why DLLMs need affix cache reuse. With intra-request caching, DLLMs still recompute the shared prefix repeatedly, with each full-sequence pass comparable to a prefill (Kedia et al., 2026), whereas an AR LLM pays for prefill once (Table 1). Across requests, AR LLMs reuse that prefix cache, while DLLMs rebuild it for each request and retain separate copies concurrently. Figure 2 previews ACache's quality–throughput tradeoff. Direct reuse offers the highest throughput in this comparison, at the cost of lower accuracy. Selecting 20%–30% of the shared tokens as request-specific Anchors, ACache exceeds $1.3\times$ full recomputation's throughput at comparable accuracy. Beyond this prefix example, shared text can occur elsewhere in the context: CacheBlend (Yao et al., 2025), EPIC (Hu et al., 2025), ProphetKV (Wang et al., 2026b), and SparseX (Zhang et al., 2026) already reuse mid-prompt text in AR inference, though their cached text still precedes generation. In a DLLM, the shared span may also follow the generation region, as in infilling-style prompts (Lou et al., 2024; Xu et al., 2026), where bidirectionality couples it to generation. To the best of our knowledge, ACache is the first cross-request cache reuse mechanism for DLLMs that is not prefix-restricted: it reuses a shared span's cache anywhere, recomputing only the part a request needs.

3 ACache Design

We formalize the DLLM inference loop (Sahoo et al., 2024). Let T denote the total number of decoding steps, $z^{(t)}$ the sequence at step t, and $\mathcal{M}_t$ the masked positions, with t descending from T to 0, so $\mathcal{M}_T$ is the whole generation region, while the tokens outside it carry the request's context. They may precede the generation region, follow it, or surround it. At each step, the model predicts all masked positions in parallel:

$$\hat{y}_i^{(t)} \sim p_\theta(\cdot \mid z^{(t)}), \quad i \in \mathcal{M}_t \tag{1}$$

Only a subset of predictions is committed. Let $\mathcal{C}_t \subseteq \mathcal{M}_t$ denote positions committed at step t. The next state is updated as:

$$z_i^{(t-1)} = \begin{cases} \hat{y}_i^{(t)}, & i \in \mathcal{C}_t \\ \texttt{[MASK]}, & i \in \mathcal{M}_t \setminus \mathcal{C}_t \\ z_i^{(t)}, & i \notin \mathcal{M}_t \end{cases} \tag{2}$$

Decoding schedules define $\mathcal{C}_t$ differently: the signal may be a confidence threshold (Nie et al., 2025b; Wu et al., 2026b; Luo et al., 2026a), an entropy bound (Ben-Hamu et al., 2025), or inter-token dependencies (Luo et al., 2026b). All, however, repeat the same "predict-then-commit" cycle.

Now consider a shared affix that occupies a contiguous span of non-masked positions $\mathcal{F} = \{l, l+1, \ldots, r\}$ in $z^{(t)}$, with length $L = |\mathcal{F}|$. Under intra-request caching mechanisms (Ma et al., 2025a; Liu et al., 2026c; Wu et al., 2026b; Jiang et al., 2026; Nguyen-Tri et al., 2026; Xiong et al., 2026), the affix in $\mathcal{F}$ is still recomputed from scratch for each request, at the start or at each of several recomputation points. ACache instead recomputes only a subset of the L affix tokens.

3.1 Anchor Selection

We refer to this subset of affix tokens as Anchor Tokens, following prior work that identifies a small critical subset of tokens in the KV cache (Zhang et al., 2023; Xiao et al., 2024). ACache selects them before the decoding loop starts and keeps that set fixed throughout decoding for each request. To make this selection request-specific while preserving affix reuse, we score affix tokens by mask-to-affix attention, a signal similar to the one MaskKV uses for cache eviction (Huang et al., 2026). Specifically,

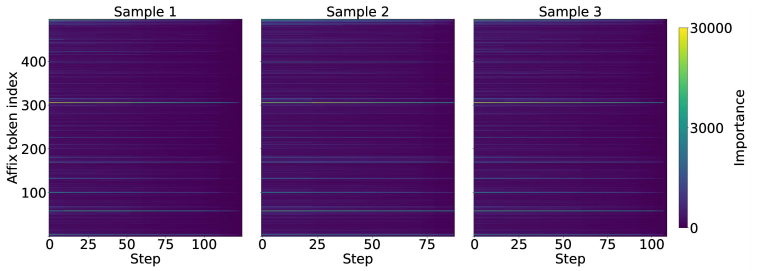


Figure 3: **Anchor Token importance.** Each panel shows a randomly selected GSM8K sample (Section 5 setup, 2-shot prompts). Color encodes mask-to-affix attention importance over affix tokens (vertical) and decoding steps (horizontal), highlighting a small subset at stable positions across steps.

we obtain this signal from a single forward pass over the non-affix tokens—the request-owned prompt and the masked generation region—with the affix contributing only its precomputed keys and values.

Formally, let $Q^{(n,h)}_{\text{mask}}$ denote the queries of masked positions at layer $n \in \{1, \dots, N\}$ and head $h \in \{1, \dots, H\}$, let $K^{(n,h)}_{\text{affix}}$ denote the precomputed keys of affix tokens, and let d_h be the per-head hidden dimension. For an affix token $j \in \mathcal{F}$ and a masked position $i \in \mathcal{M}_T$, we denote by $a^{(n,h)}_{i,j}$ the attention weight that masked query i places on affix key j, normalized over the affix keys alone:

$$a^{(n,h)}_{i,j} = \Big[\mathrm{Softmax}\Big(\frac{Q^{(n,h)}_{\text{mask}}(K^{(n,h)}_{\text{affix}})^\top}{\sqrt{d_h}}\Big)\Big]_{i,j} \tag{3}$$

Summing these weights over all layers, heads, and initial masked positions gives the importance score of affix token j:

$$s_j = \sum_{n=1}^{N}\sum_{h=1}^{H}\sum_{i\in\mathcal{M}_T} a^{(n,h)}_{i,j} \tag{4}$$

ACache then takes the positions of the top-K scores as the Anchor set:

$$\mathcal{A} = \mathrm{TopK}\big(\{s_j\}_{j\in\mathcal{F}}, K\big), \qquad \mathcal{S} = \mathcal{F} \setminus \mathcal{A} \tag{5}$$

Here $\mathcal{S}$ denotes the remaining non-anchor affix tokens. We set $K = \lceil \rho L \rceil$ for a fixed *Anchor ratio* ρ, which is the only hyperparameter of ACache; Section 5.1 sweeps this ratio.

Because importance concentrates on a small subset, recomputing only that subset preserves context consistency; because a selection made at request start captures nearly as much importance as one made later, one selection per request suffices, as in prior work (Liu et al., 2023; Li et al., 2024; Jiang et al., 2026; Huang et al., 2026). Figure 3 shows both; Appendix D.1 measures them over the full test sets. That selection costs one extra forward pass, cheaper than a full one, against the T decoding steps. Within it, the mask-to-affix aggregation costs $O(NHmL)$, where $m = |\mathcal{M}_T|$; top-K extraction over the L scores then adds $O(L \log K)$. Appendix D.2 confirms these costs are small.

3.2 Selective Recomputation

The shared affix KV cache is precomputed once from a forward pass over the affix span alone, and reused across every request whose context contains that affix; each request selects its Anchor set $\mathcal{A}$ at request start. At all recomputation points, including request start, ACache recomputes the KV states for Anchor Tokens and current request-owned tokens, while tokens in $\mathcal{S}$ continue to use the shared cache. Any decoding schedule can proceed unchanged on top of this cache layout. In this way, the recomputation cost on a shared affix drops from $O(L)$ to $O(K)$, and since K is typically much smaller than L, ACache preserves most of the efficiency benefit of affix reuse. At the same time, the Anchor Tokens remain synchronized with the evolving request-specific context.

4 ACache System Support

To make ACache's affix cache reuse pay off, the runtime can rely on none of the three assumptions behind AR prefix caching: every request reuses the whole shared span; every request finds it at

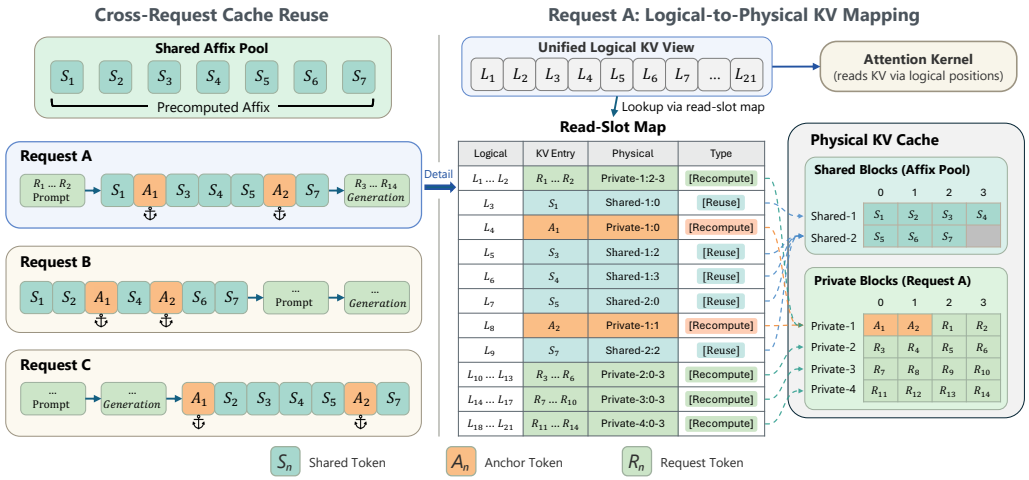


Figure 4: **System architecture of ACache.** *Left*: requests reuse one precomputed affix cache as an infix (A), prefix (B), or suffix (C), selecting request-specific Anchor Tokens. *Right*: one request's read-slot map, resolving each logical position to a private or shared KV slot.

offset zero; and no request rewrites its KV once prefill ends. We build that runtime on top of Nano-vLLM (Nano-vLLM, 2025), a lightweight version of the vLLM engine (Kwon et al., 2023), layering ACache's cache management and admission logic over intra-request caching mechanisms.

4.1 KV Layout and Read-Slot Map

The system partitions the paged KV cache into reserved shared blocks and the request-private blocks outside them. The reserved blocks hold the precomputed affix KV cache, stored once rather than per request; each request's private blocks then store the KV of two groups: the Anchor Tokens selected from that affix, and the request-owned prompt and generation-region tokens. Figure 4 shows shared affix cache reuse across requests on the left, and one request's layout on the right.

A simpler design would materialize a blended cache per request: copy the shared affix cache, patch in the recomputed Anchor KV, and run standard attention. Selective-recompute systems for AR inference can afford exactly that (Yao et al., 2025; Hu et al., 2025; Wang et al., 2026b; Zhang et al., 2026), because the blend duplicates nothing that concurrent requests were sharing in memory. ACache's affix cache is shared in memory by every request that uses it: copying fills each request's private blocks with the whole affix cache rather than the Anchors alone, so fewer requests fit in the KV cache concurrently, and restricting the copy to the blocks the Anchors land in barely helps, since the selection criterion scatters Anchors across blocks that hold hundreds of token slots each. In a DLLM, mixed ownership is also the steady state rather than a transient: AR inference freezes context KV at the end of prefill, but Anchor and request-owned KV is rewritten at every recomputation point.

ACache therefore composes shared and private KV in the read path. A per-request *read-slot map* (Figure 4, right) binds every logical position to the physical KV slot to read from, sending Anchors to private slots and the rest of the affix to shared ones, so attention sees one logical full-context view without a blended copy. Appendix B.1 details the kernel built around this map (Algorithm 2).

4.2 Shared-Affix Cache Rotation

The shared affix cache is stored after positional rotation, as though every affix began at position zero. Rotary embeddings compose additively—rotating by two offsets in sequence equals rotating once by their sum (Su et al., 2024)—so that one copy can be used at any starting offset with no approximation (Figure 4, left). SparseX exploits the same property by re-basing the stored keys, which costs a rotated copy per request (Zhang et al., 2026). ACache instead keeps the correction on the query side: the kernel counter-rotates the query by the request's offset before scoring it against shared slots, so the shared cache is never rewritten and serves requests at different offsets concurrently. Affixes already at position zero skip the correction altogether. Appendix B.1 details this correction.

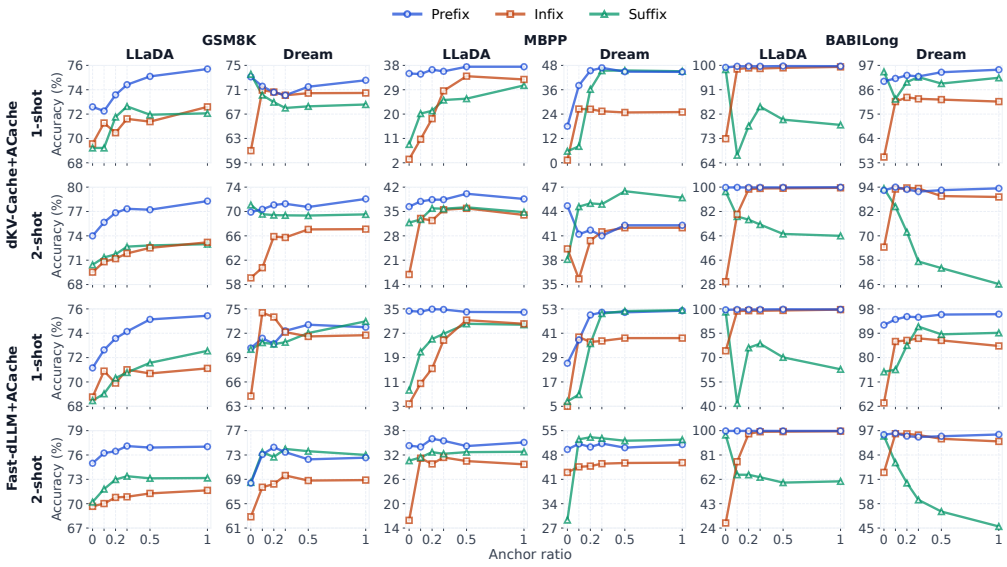


Figure 5: **Accuracy with ACache.** Accuracy increases with the Anchor ratio (0: direct reuse; 1: full recomputation), except BABILong suffix, which declines overall.

4.3 Admission and Inference Loop

On admission, the runtime selects a request's Anchor Tokens by mask-to-affix importance and allocates private slots whose positions start out awaiting recomputation (Algorithm 1, Admit). During KV recomputation, the runtime forms queries only for those positions, reading keys and values through the read-slot map. The full-context view seen by attention is unchanged, but the query work shrinks, as Section 5.2 quantifies. During decoding, queries are instead formed only for the request's currently active region: the resulting KV states are written to the private slots already assigned by the read-slot map, and attention again reads the full-context view through the same map.

Algorithm 1 ACache request lifecycle

procedure Admit(r)
  Reuse r's shared affix cache
  Select r's Anchor Tokens
  Allocate private slots for r's Anchor and request-owned tokens
  Build r's read-slot map
end procedure
procedure InferenceLoop
  while requests remain **do**
    Advance or finish sequences
    if any sequence just finished **then**
      Admit(r) for each newly admissible r
    end if
    $R \leftarrow$ sequences that need recomputation
    if $R \neq \emptyset$ **then**
      Recompute KV states at each $r \in R$'s positions awaiting recomputation, via its read-slot map
      continue ▷ skip decode this step
    else
      Decode one step for the active sequences
    end if
  end while
end procedure

Recomputation points are per-request and unsynchronized, so sequences in a batch reach them at different steps; that is what makes it worth scheduling at step granularity. Algorithm 1 (InferenceLoop) puts recomputation ahead of the next decode step, a continuous batching pattern that lets a request finishing recomputation join the very next decode batch, in the spirit of ORCA and vLLM-Omni (Yu et al., 2022; vLLM-Omni, 2026). Appendix B.2 describes how each underlying caching mechanism refills the batch around this loop.

5 Evaluation

We test the three claims ACache's case rests on: that recomputing a small affix subset restores most of the accuracy direct reuse loses; that the recomputation it saves survives into end-to-end gains; and that its choice of what to recompute holds up against the closest baselines at a matched budget.

Table 2: **LLaDA recompute latency (ms) under prefix reuse.** Dream: Table 9.

Dataset	Batch	1-shot		2-shot		4-shot	
		Fast-dLLM	ACache	Fast-dLLM	ACache	Fast-dLLM	ACache
GSM8K	1	56.3	38.9 (30.8% ↓)	91.2	46.1 (49.5% ↓)	124.7	60.0 (51.9% ↓)
	4	62.8	43.5 (30.7% ↓)	100.6	52.3 (48.0% ↓)	144.1	65.5 (54.5% ↓)
	16	87.9	61.9 (29.5% ↓)	147.2	76.9 (47.8% ↓)	213.9	92.6 (56.7% ↓)
MBPP	1	58.2	44.3 (23.8% ↓)	91.7	50.6 (44.8% ↓)	133.0	65.7 (50.6% ↓)
	4	61.7	47.1 (23.6% ↓)	96.8	54.4 (43.8% ↓)	145.9	69.9 (52.1% ↓)
	16	75.1	59.2 (21.1% ↓)	112.2	65.2 (41.9% ↓)	176.6	82.8 (53.1% ↓)

Table 3: **End-to-end throughput under prefix reuse.** Measured in generated tokens per second.

Model	Dataset	Batch	1-shot		2-shot		4-shot	
			Fast-dLLM	ACache	Fast-dLLM	ACache	Fast-dLLM	ACache
LLaDA	GSM8K	1	104	99 (0.96×)	94	98 (1.04×)	85	92 (1.09×)
		4	263	274 (1.04×)	206	253 (1.23×)	166	229 (1.37×)
		16	394	433 (1.10×)	278	378 (1.36×)	204	329 (1.61×)
	MBPP	1	209	213 (1.02×)	168	194 (1.16×)	123	155 (1.26×)
		4	380	401 (1.06×)	272	366 (1.34×)	189	289 (1.53×)
		16	472	507 (1.07×)	325	446 (1.37×)	210	353 (1.68×)
Dream	GSM8K	1	129	125 (0.97×)	120	120 (1.00×)	103	117 (1.13×)
		4	311	322 (1.04×)	255	298 (1.17×)	195	276 (1.42×)
		16	444	486 (1.09×)	323	434 (1.34×)	231	376 (1.63×)
	MBPP	1	195	190 (0.98×)	175	183 (1.05×)	137	171 (1.25×)
		4	402	410 (1.02×)	320	381 (1.19×)	217	331 (1.53×)
		16	520	553 (1.06×)	392	504 (1.29×)	246	421 (1.71×)

We describe the experimental setup as follows:

(i) **Datasets.** We evaluate math reasoning on GSM8K (Cobbe et al., 2021), code generation on MBPP (Austin et al., 2021b), and context retrieval on BABILong-`0k/qa1` (Kuratov et al., 2024).

(ii) **Models.** Following prior work (Kong et al., 2025; Liu et al., 2026c; Ma et al., 2025a; Song et al., 2026; Jiang et al., 2026; Luo et al., 2026a;b), we evaluate two DLLMs: LLaDA-8B-Instruct (LLaDA) (Nie et al., 2025b) and Dream-v0-Instruct-7B (Dream) (Ye et al., 2025a).

(iii) **Hardware.** We conduct all experiments on NVIDIA A100-SXM4-40GB GPUs.

(iv) **Frameworks.** Quality experiments use Hugging Face Transformers; efficiency experiments use the system described in Section 4. Both place affixes at any position exactly (Appendix C.1), so the stale-cache effect studied below is the only approximation.

(v) **Default settings.** We use a generation length of 256 and a block length of 32 tokens, following prior work (Jiang et al., 2026; Xiao et al., 2026; Luo et al., 2026a;b; Kasa et al., 2026; Lee et al., 2026a), and 8 tokens for both on BABILong, to accommodate its short answers. Parallel decoding uses Fast-dLLM's confidence threshold of 0.9 (Wu et al., 2026b). Quality evaluation sweeps Anchor ratios in $\{0, 0.1, 0.2, 0.3, 0.5, 1.0\}$. All results are averaged over two runs.

(vi) **Affix construction.** For each task and shot count, we place the same examples before the query (prefix), between the query header and answer prompt (infix), or after the masked generation region (suffix). Appendix C.2 gives the templates; Appendix D.8 reuses a system prompt.

5.1 Accuracy Under Shared Affixes

Figure 5 plots accuracy as the Anchor ratio varies from 0 (direct reuse) to 1 (full recomputation) for prefix, infix, and suffix reuse. We omit 4-shot affixes, the longest we use, from the figure: in our runs, they match full recomputation at a lower Anchor ratio than the 1- and 2-shot settings shown. Direct reuse is often inaccurate, confirming that a DLLM KV cache for shared text needs request-specific adaptation to remain accurate across requests. Affix position also affects accuracy, consistent with the finding that query position is a first-order variable in bidirectional DLLM decoding (Li et al., 2026). BABILong suffix goes further: accuracy declines overall as recomputation increases, unlike every other position-dataset pair. We attribute this reversal to BABILong's answer format: the answer is a bare location, and the few-shot examples end in locations of their own; under suffix placement these sit right next to the masked generation and can interfere with the model's answer. We exclude it from the averages below, which would otherwise count the reversal as recovered accuracy.

Table 4: **LLaDA peak KV cache memory usage (GB) under prefix reuse.** Dream: Table 10.

Dataset	Batch	1-shot		2-shot		4-shot	
		Fast-dLLM	ACache	Fast-dLLM	ACache	Fast-dLLM	ACache
GSM8K	1	0.38	0.38 (0.0%)	0.50	0.75 (50.0% ↑)	0.75	0.88 (16.7% ↑)
	4	1.50	1.12 (25.0% ↓)	2.00	1.50 (25.0% ↓)	3.00	2.00 (33.3% ↓)
	16	6.00	4.12 (31.2% ↓)	8.00	4.62 (42.2% ↓)	12.00	6.50 (45.8% ↓)
MBPP	1	2.12	2.12 (0.0%)	2.25	2.25 (0.0%)	2.50	2.62 (5.0% ↑)
	4	3.25	3.00 (7.7% ↓)	3.75	3.25 (13.3% ↓)	4.75	3.75 (21.1% ↓)
	16	8.12	6.25 (23.1% ↓)	10.12	6.50 (35.8% ↓)	13.75	8.25 (40.0% ↓)

The remaining loss depends on dataset and position jointly: averaged over models, shot counts, and frameworks, direct reuse costs 1.73–5.25 points on GSM8K across positions, while MBPP rises from 7.10 at prefix to 18.68 at infix and 20.45 at suffix, and BABILong from 1.40 at prefix to 35.03 at infix. As the ratio increases, ACache restores accuracy across settings; ratio 0.2 recovers about 87% of the accuracy lost to direct reuse. By shot count, ratio 0.2 raises 1-shot accuracy from 51.11% to 62.21%, vs. 64.88% of full recomputation; in 2-shot settings direct reuse starts higher at 56.96%, and ratio 0.2 nearly matches full recomputation (66.04% vs. 66.28%). Few-shot examples become more redundant, so accuracy becomes less sensitive to the Anchor ratio and to the selection criterion.

5.2 System Efficiency

Here we fix the Anchor ratio at 0.2, which on average recovers most of the lost accuracy at 1 shot and matches full recomputation at 2 and 4 shots. We use Fast-dLLM, the more efficient of the two caching frameworks; Appendix D.4 reports dKV-Cache. We focus on prefix reuse; Appendix D.5 reports infix and suffix, with similar results. We omit BABILong for its short generation length. Table 2 isolates ACache's direct effect on LLaDA recomputation. ACache reduces recompute latency in every setting, by 21.1% to 56.7%. Table 3 reports end-to-end throughput. At batch size 1 with the shortest, 1-shot affix, throughput ranges from $0.96\times$ to $1.02\times$: Anchor selection overhead is least amortized there, and Appendix D.2 breaks it down. We include batch size 1 to bound that cost, though it is not the regime a cross-request mechanism targets. Recomputation savings outweigh the overhead in every other setting, reaching up to $1.71\times$ at 4-shot, where ratio 0.2 matches full recomputation.

Table 4 reports LLaDA peak KV cache memory for the same settings. ACache reduces peak KV cache usage as batch size and shared-affix length increase, reaching the largest reductions at batch size 16 and 4-shot: on GSM8K from 12.00 GB to 6.50 GB, a 45.8% drop, and on MBPP from 13.75 GB to 8.25 GB, a 40.0% drop. These savings come from the read-slot map, which keeps one copy of the affix cache where the baseline, like a per-request blended copy (Section 4), keeps one per request, leaving headroom for more concurrent requests. At batch size 1, ACache uses up to 0.25 GB more, as shared and private blocks are allocated separately at block granularity.

5.3 Comparison to Closest Baselines

None of the three systems closest to ACache applies directly to DLLM affix reuse. CacheBlend (Yao et al., 2025) and ProphetKV (Wang et al., 2026b) are autoregressive: both recompute once during prefill and overwrite the selected entries in a per-request copy of the cache. A DLLM rewrites affix KV at every recomputation point, so it would repeat that overwrite throughout decoding, each request holding its own copy of the shared affix cache (Section 4). BiCache (Go et al., 2026) targets DLLMs, but only shared prefixes, and reads reuse depth from an offline profile rather than a budget we can match. We therefore compare criteria, not whole systems, porting each into ACache's pipeline at a matched Anchor ratio ρ. CacheBlend scores tokens by KV deviation, ProphetKV by request-prompt attention, and BiCache recomputes the affix in the last $\lceil \rho N \rceil$ of N layers (Appendix C.3).

Table 5 reports this comparison at Anchor ratios 0.2 and 0.3 in the 1-shot setting, where accuracy is most sensitive to the selection criterion (Section 5.1). ACache leads the next-best criterion, ProphetKV, by 2.0 to 2.3 points of average accuracy on Dream and matches it on LLaDA, within 0.1 points; it leads CacheBlend, likewise token-level, by 3.7 to 9.4. Which tokens are recomputed thus matters at a fixed budget, and mask-to-affix attention importance is a well-suited criterion for DLLM affix reuse, with the advantage most visible on infix and suffix, where direct reuse costs the most accuracy.

Table 5: **ACache's criterion compared to adapted baselines, 1-shot.** Accuracy (%) on Fast-dLLM. Bold marks the best value per column; underline, the second-best. BiCache† is the official implementation; Ratio is the mean equivalent Anchor ratio. Avg. omits BABILong suffix (Section 5.1).

Model	Ratio	Criterion	GSM8K			MBPP			BABILong			Avg.
			Pfx.	Inf.	Suf.	Pfx.	Inf.	Suf.	Pfx.	Inf.	Suf.	
LLaDA	0.2	CacheBlend	71.6	69.6	69.8	**35.5**	7.8	22.0	99.5	82.8	63.6	57.3
		ProphetKV	**75.6**	69.8	**70.8**	33.5	13.1	**25.8**	**99.7**	98.6	43.4	60.9
		BiCache	71.4	69.1	68.2	33.2	3.4	8.4	99.4	73.9	**98.4**	53.4
		ACache	73.6	**69.9**	70.3	34.9	**15.4**	25.1	99.6	**98.9**	76.1	**61.0**
	0.3	CacheBlend	71.9	70.4	70.9	**35.6**	12.3	15.2	99.4	94.7	27.8	58.8
		ProphetKV	**75.3**	70.5	**72.1**	34.9	23.0	**27.0**	99.4	98.8	43.8	**62.6**
		BiCache	71.0	67.9	67.6	33.8	3.1	10.3	99.4	68.1	**98.2**	52.6
		ACache	74.1	**71.0**	70.8	34.8	**24.7**	26.8	**99.7**	**98.9**	78.6	**62.6**
	0.67	BiCache†	76.1	–	–	34.6	–	–	99.5	–	–	–
Dream	0.2	CacheBlend	**73.4**	68.2	**73.5**	38.6	4.8	4.9	**96.4**	84.3	80.5	55.5
		ProphetKV	70.4	73.5	69.3	49.9	32.3	27.4	95.9	82.3	**91.5**	62.6
		BiCache	70.9	64.3	71.4	26.8	4.8	7.0	92.1	55.0	77.2	49.0
		ACache	70.7	**74.0**	70.7	**50.0**	**36.7**	**35.9**	95.2	**86.4**	84.5	**64.9**
	0.3	CacheBlend	73.5	71.1	**73.6**	**52.2**	14.4	18.0	**95.6**	**87.1**	86.3	60.7
		ProphetKV	72.2	71.6	71.1	49.8	**37.7**	39.3	95.3	83.4	83.4	65.0
		BiCache	**73.7**	66.1	70.5	41.1	3.5	9.6	90.6	45.6	77.5	50.1
		ACache	72.3	**72.1**	70.9	51.3	37.2	**50.7**	94.9	**87.1**	**91.5**	**67.1**
	0.77	BiCache†	71.4	–	–	51.9	–	–	91.0	–	–	–

Table 6: **End-to-end throughput at batch size 1 under prefix reuse (tokens/s).** Both run on Fast-dLLM: BiCache from its official implementation (Appendix C.3), ACache at Anchor ratio 0.2.

Model	Dataset	1-shot		2-shot		4-shot	
		BiCache	ACache	BiCache	ACache	BiCache	ACache
LLaDA	GSM8K	96	99 (1.03×)	84	98 (1.18×)	71	92 (1.30×)
	MBPP	192	213 (1.11×)	165	194 (1.18×)	118	155 (1.31×)
Dream	GSM8K	87	125 (1.44×)	93	120 (1.30×)	83	117 (1.41×)
	MBPP	145	190 (1.31×)	136	183 (1.35×)	119	171 (1.44×)

Appendix D.7 further swaps ACache's ranking for random and bottom selection. BiCache places last on both models, at both ratios. Averaged over the three prefix settings, its official implementation reaches 70.1% on LLaDA and 71.4% on Dream, against ACache's 69.4% and 72.0% at ratio 0.2, but at a mean equivalent Anchor ratio of 0.67 and 0.77. Whole layers thus seem a coarser unit than tokens for preserving request-specific information, though a criterion could select on both.

For system efficiency, the four matched criteria all spend the same recomputation budget on every request that shares an affix. BiCache's profiled depth, as originally designed, instead varies with each request's shared-prefix fraction, so recomputation cost differs across a batch—which is hard for a continuous batching loop to absorb (Yu et al., 2022). Table 6 compares official BiCache with ACache at batch size 1, the only one BiCache's open-source engine supports: ACache is faster in all twelve settings, by $1.03\times$ to $1.44\times$. There the advantage comes instead from the recomputation ACache saves and from more efficient system support; ACache also batches requests, widening the gap.

6 Conclusion

We presented ACache, in which one Anchor ratio governs how much of a shared span's KV cache a DLLM reuses, and one copy serves every request whether it places that span as a prefix, infix, or suffix. Mask-to-affix attention importance concentrates on a small, stable subset of affix tokens, so recomputing around 20% of them—the Anchor Tokens—recovers most of the accuracy direct reuse loses, and at a matched budget this criterion selects better than those adapted from the closest baselines. The recomputation and memory ACache saves grow with both batch size and shared-span length, so the gains are largest in exactly the regime cross-request reuse targets. In the future, we hope the affix can do for DLLM inference stacks what cross-request reuse has done for AR ones.

AI USE STATEMENT

In this work, we used generative AI tools to assist with implementing parts of the codebase, including evaluation scripts. We have not used generative AI tools to propose or refine hypotheses, design research methodology or experiments. Additionally, we used generative AI tools to assist with polishing the manuscript for wording, grammar, and clarity, and to support literature search during related-work review. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We describe the ACache design and Anchor Token selection procedure in Section 3, and the system design and implementation details in Section 4 and Appendix B. Evaluation settings are reported under "Default settings" in Section 5; exact prompt templates for each affix position are given in Appendix C.2. All datasets (GSM8K, MBPP, BABILong) and base models (LLaDA, Dream) are publicly available, and all measurements were taken on the GPUs reported under "Hardware" in Section 5. Because ACache adds a custom Triton attention kernel, the code pins its package versions and documents the PyTorch, CUDA, and Triton installation it expects. The ACache implementation, including the Nano-vLLM-based system and the scripts used to reproduce our experiments, is provided as anonymized supplementary material with this submission; a link to the public repository will be included after the review procedure.

REFERENCES

Marianne Arriola, Subham Sekhar Sahoo, Aaron Gokaslan, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Justin T Chiu, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=tyEyYT267x.

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NeurIPS '21, Red Hook, NY, USA, 2021a. Curran Associates Inc. ISBN 9781713845393.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021b.

Heli Ben-Hamu, Itai Gat, Daniel Severo, Niklas Nolte, and Brian Karrer. Accelerated sampling from masked diffusion models via entropy bounded unmasking. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=WBcBhT1NKO.

Tiwei Bie, Maosong Cao, Kun Chen, Lun Du, Mingliang Gong, Zhuochen Gong, Yanmei Gu, Jiaqi Hu, Zenan Huang, Zhenzhong Lan, et al. LLaDA2.0: Scaling up diffusion language models to 100B. *arXiv preprint arXiv:2512.15745*, 2025.

Xinhua Chen, Sitao Huang, Cong Guo, Chiyue Wei, Yintao He, Jianyi Zhang, Hai Helen Li, and Yiran Chen. DPad: Efficient diffusion language models with suffix dropout. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=0yOsSMU1eY.

Shuang Cheng, Yihan Bian, Dawei Liu, Yuhua Jiang, Yihao Liu, Linfeng Zhang, Qian Yao, Zhongbo Tian, Wenhai Wang, Qipeng Guo, Kai Chen, Biqing Qi, and Bowen Zhou. SDAR: A synergistic diffusion-AutoRegression paradigm for scalable sequence generation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 22058–22075, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1110. URL https://aclanthology.org/2026.findings-acl.1110/.

Minsoo Cheong, Donghyun Son, Woosang Lim, and Sungjoo Yoo. EntropyCache: Decoded token entropy guided KV caching for diffusion language models. *arXiv preprint arXiv:2603.18489*, 2026.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: fast and memory-efficient exact attention with io-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

DiffusionGemma Team, Adrien Ali Taïga, James Assiene, Daniele Calandriello, Rahma Chaabouni, João Gante, Tamara von Glehn, Nate Keating, Chris Knutsen, Martin Kukla, et al. Diffusiongemma technical report. *arXiv preprint arXiv:2608.00146*, 2026.

Jiakun Fan, Yanglin Zhang, Xiangchen Li, and Dimitrios S Nikolopoulos. Taming the memory footprint crisis: System design for production diffusion llm serving. *arXiv preprint arXiv:2512.17077*, 2025.

Natalia Frumkin, Bokun Wang, Hung-Yueh Chiang, Chi-Chih Chang, Mohamed S Abdelfattah, and Diana Marculescu. Dare: Diffusion language model activation reuse for efficient inference. *arXiv preprint arXiv:2605.08134*, 2026.

Yonggan Fu, Lexington Whalen, Zhifan Ye, Xin Dong, Shizhe Diao, Jingyu Liu, Chengyue Wu, Hao Zhang, Enze Xie, Song Han, Maksim Khadkevich, Jan Kautz, Yingyan Celine Lin, and Pavlo Molchanov. Efficient-DLM: From autoregressive to diffusion language models, and beyond in speed. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=ynRnOqI2W8.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL https://zenodo.org/records/12608602.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt Cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

Younghun Go, Jaehoon Han, Changyong Shin, Chuck Yoo, and Gyeongsik Yang. Enabling kv caching of shared prefix for diffusion language models. *arXiv preprint arXiv:2606.07571*, 2026.

Shansan Gong, Shivam Agarwal, Yizhe Zhang, Jiacheng Ye, Lin Zheng, Mukai Li, Chenxin An, Peilin Zhao, Wei Bi, Jiawei Han, Hao Peng, and Lingpeng Kong. Scaling diffusion language models via adaptation from autoregressive models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=j1tSLYKwg8.

Xiaochuang Han, Sachin Kumar, and Yulia Tsvetkov. SSD-LM: Semi-autoregressive simplex-based diffusion language model for text generation and modular control. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11575–11596, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.647. URL https://aclanthology.org/2023.acl-long.647/.

Andre He, Sean Welleck, and Daniel Fried. Reasoning with latent tokens in diffusion language models. *arXiv preprint arXiv:2602.03769*, 2026a.

Xuning He, Zinan Sheng, Yongding Tao, Huanyu Liu, Ge Li, Xue Jiang, and Yihong Dong. Archer: Adaptive reuse of cached hidden states for efficient rollback in diffusion language models. *arXiv preprint arXiv:2608.08086*, 2026b.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

Emiel Hoogeboom, Didrik Nielsen, Priyank Jaini, Patrick Forré, and Max Welling. Argmax flows and multinomial diffusion: learning categorical distributions. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc. ISBN 9781713845393.

Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, tiancheng hu, zhang qin, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. EPIC: Efficient position-independent caching for serving large language models. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=qjd3ZUiHRT.

Zhanqiu Hu, Jian Meng, Yash Akhauri, Mohamed S. Abdelfattah, Jae sun Seo, Zhiru Zhang, and Udit Gupta. FlashDLM: Accelerating diffusion language model inference via efficient KV caching and guided diffusion. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=KUfKvlX3VY.

Jianuo Huang, Yaojie Zhang, Yicun Yang, Benhao Huang, and Linfeng Zhang. Mask tokens as prophet: Fine-grained cache eviction for efficient dLLM inference. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 3456–3479, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.170. URL https://aclanthology.org/2026.findings-acl.170/.

Inception Labs, Samar Khanna, Siddhant Kharbanda, Shufan Li, Harshit Varma, Eric Wang, Sawyer Birnbaum, Ziyang Luo, Yanis Miraoui, Akash Palrecha, et al. Mercury: Ultra-fast language models based on diffusion. *arXiv preprint arXiv:2506.17298*, 2025.

Daniel Mingyi Israel, Guy Van den Broeck, and Aditya Grover. Accelerating diffusion LLMs via adaptive parallel decoding. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=xwqTt26NJf.

Yuchu Jiang, Yue Cai, Xiangzhong Luo, Jiale Fu, Jiarui Wang, Chonghan Liu, and Xu Yang. d^{2}cache: Accelerating diffusion-based LLMs via dual adaptive caching. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=SjInfpK5RM.

Jordan Juravsky, Bradley Brown, Ryan Saul Ehrlich, Daniel Y Fu, Christopher Re, and Azalia Mirhoseini. Hydragen: High-throughput LLM inference with shared prefixes. In *Workshop on Efficient Systems for Foundation Models II @ ICML2024*, 2024. URL https://openreview.net/forum?id=Ye3ia34Fpp.

Siva Rajesh Kasa, Yasong Dai, Sumit Negi, and Hongdong Li. Fast-dllm++: Fréchet profile decoding for faster diffusion llm inference. *arXiv preprint arXiv:2606.02955*, 2026.

Nitin Kedia, Saurabh Agarwal, Myungjin Lee, and Aditya Akella. Sangam: Efficiently serving diffusion llms with the ar stack. *arXiv preprint arXiv:2607.04206*, 2026.

Fanheng Kong, Jingyuan Zhang, Yahui Liu, Zirui Wu, Yu Tian, Guorui Zhou, et al. Accelerating diffusion llm inference via local determinism propagation. *arXiv preprint arXiv:2510.07081*, 2025.

Yuri Kuratov, Aydar Bulatov, Petr Anokhin, Ivan Rodkin, Dmitry Sorokin, Artyom Sorokin, and Mikhail Burtsev. BABILong: Testing the limits of llms with long context reasoning-in-a-haystack. *Advances in Neural Information Processing Systems*, 37:106519–106554, 2024.

Omin Kwon, Doyeon Kim, Jongseok Park, Seung Yul Lee, Ion Stoica, and Jae W Lee. HERALD: High-throughput block diffusion llm serving via cpu-gpu cooperative kv cache retrieval. *arXiv preprint arXiv:2606.21633*, 2026.

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*, pp. 611–626, 2023.

Mingyu Lee, Akshat Ramachandran, Souvik Kundu, and Tushar Krishna. Polestar: Drift-aware cache calibration and token commitment for efficient inference of diffusion llms. *arXiv preprint arXiv:2607.14107*, 2026a.

Younjoo Lee, Seungkyun Dan, Junghoo Lee, Jaiyoung Park, and Jung Ho Ahn. DyLLM: Efficient diffusion LLM inference via saliency-based token selection and partial attention. In *Forty-third International Conference on Machine Learning*, 2026b. URL https://openreview.net/forum?id=0azUrmsSyA.

Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori B. Hashimoto. Diffusion-LM improves controllable text generation. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NeurIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 22947–22970. Curran Associates, Inc., 2024. doi: 10.52202/079017-0722. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/28ab418242603e0f7323e54185d19bde-Paper-Conference.pdf.

Zhengheng Li, Panrui Li, Xuyang Liu, and Puzhi Xia. Where to place the query? unveiling and mitigating positional bias in in-context learning for diffusion llms via decoding dynamics. In *International Conference on Intelligent Computing*, pp. 602–612. Springer, 2026.

Kaihua Liang, Xin Tan, An Zhong, Hong Xu, and Marco Canini. FOCUS: DLLMs know how to tame their compute bound. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=40fUEdwvH3.

Song Liu, Yan Liu, Jinhua Cui, Shiqiang Nie, Jinyu Wang, Weiguo Wu, et al. Adaptive kv cache reuse for fast long-context llm serving. *arXiv preprint arXiv:2605.24022*, 2026a.

Yang Liu, Zhaokai Luo, Huayi Jin, Zhiyong Wang, Ruozhou He, Boyu Wang, Guanjie Chen, Tao Xie, and Junhao Hu. RedKnot: Efficient long-context llm serving with head-aware kv reuse and segpagedattention. *arXiv preprint arXiv:2606.06256*, 2026b.

Zhiyuan Liu, Yicun Yang, Yaojie Zhang, Junjie Chen, Chang Zou, Qingyan Wei, Shaobo Wang, Yichen Zhu, and Linfeng Zhang. dLLM-cache: Accelerating diffusion large language models with adaptive caching. In *Forty-third International Conference on Machine Learning*, 2026c. URL https://openreview.net/forum?id=guLrgDb2jB.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 52342–52364. Curran Associates, Inc., 2023. doi: 10.52202/075280-2279. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/a452a7c6c463e4ae8fbdc614c6e983e6-Paper-Conference.pdf.

Lingkun Long, Yushi Huang, Shihao Bai, Ruihao Gong, Jun Zhang, Ao Zhou, and Jianlei Yang. FocusdLLM: Accelerating long-context diffusion LLM inference via confidence-guided context focusing. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12146–12160, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.556. URL https://aclanthology.org/2026.acl-long.556/.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Guanxi Lu, Hao Mark Chen, Yuto Karashima, Zhican Wang, Daichi Fujiki, and Hongxiang Fan. AdaBlock-dLLM: Semantic-aware diffusion LLM inference via adaptive block size. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=0Cv9PwL7cI.

Lizhuo Luo, Shenggui Li, Yonggang Wen, and Tianwei Zhang. DSB: Dynamic sliding block scheduling for diffusion llms. *arXiv preprint arXiv:2602.05992*, 2026a.

Lizhuo Luo, Zhuoran Shi, Jiajun Luo, Zhi Wang, Shen Ren, Wenya Wang, and Tianwei Zhang. DAWN: Dependency-aware fast inference for diffusion llms. *arXiv preprint arXiv:2602.06953*, 2026b.

Xinyin Ma, Runpeng Yu, Gongfan Fang, and Xinchao Wang. dKV-Cache: The cache for diffusion language models. In *Advances in Neural Information Processing Systems*, 2025a.

Yuxin Ma, Lun Du, Lanning Wei, Kun Chen, Qian Xu, Kangyu Wang, Guofeng Feng, Guoshan Lu, Lin Liu, Xiaojing Qi, et al. dInfer: An efficient inference framework for diffusion language models. *arXiv preprint arXiv:2510.08666*, 2025b.

Nano-vLLM. Nano-vLLM: A lightweight implementation of vLLM. https://github.com/GeeekExplorer/Nano-vLLM, 2025. GitHub repository.

Quan Nguyen-Tri, Mukul Ranjan, and Zhiqiang Shen. Attention is all you need for KV cache in diffusion LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=zkUbhdAiFJ.

Jinjie Ni, Qian Liu, Longxu Dou, Chao Du, Zili Wang, Hang Yan, Tianyu Pang, and Michael Qizhe Shieh. Diffusion language models are super data learners. *arXiv preprint arXiv:2511.03276*, 2025.

Shen Nie, Fengqi Zhu, Chao Du, Tianyu Pang, Qian Liu, Guangtao Zeng, Min Lin, and Chongxuan Li. Scaling up masked diffusion models on text. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=WNvvwK0tut.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. In *Advances in Neural Information Processing Systems*, 2025b.

Shen Nie, Qiyang Min, Shaoxuan Xu, Zihao Huang, Yuxuan Song, Yong Shan, Yankai Lin, Wayne Xin Zhao, Chongxuan Li, and Ji-Rong Wen. Improved large language diffusion models. *arXiv preprint arXiv:2606.25331*, 2026.

Jingyang Ou, Shen Nie, Kaiwen Xue, Fengqi Zhu, Jiacheng Sun, Zhenguo Li, and Chongxuan Li. Your absorbing discrete diffusion secretly models the conditional distributions of clean data. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=sMyXP8Tanm.

Bei Ouyang, Yifan Qiao, and Jiarong Xing. Unicache: Unifying prefix cache eviction for heterogeneous llm serving workloads. *Proc. ACM Meas. Anal. Comput. Syst.*, 10(2), May 2026. doi: 10.1145/3805652. URL https://doi.org/10.1145/3805652.

Zaifeng Pan, Yitong Ding, Yue Guan, Zheng Wang, Zhongkai Yu, Xulong Tang, Yida Wang, and Yufei Ding. FastTree: Optimizing attention kernel and runtime for tree-structured llm inference. In M. Zaharia, G. Joshi, and Y. Lin (eds.), *Proceedings of Machine Learning and Systems*, volume 7. MLSys, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/file/96894468eb44631a32d7ebd56f9892c7-Paper-Conference.pdf.

Jianhui Pang, Fanghua Ye, Derek Wong, Xin He, Wanshun Chen, and Longyue Wang. Anchor-based large language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 4958–4976, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.295. URL https://aclanthology.org/2024.findings-acl.295/.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In D. Song, M. Carbin, and T. Chen (eds.), *Proceedings of Machine Learning and Systems*, volume 5, pp. 606–624. Curan, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/file/c4be71ab8d24cdfb45e3d06dbfca2780-Paper-mlsys2023.pdf.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pp. 155–170, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/qin.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NeurIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis Titsias. Simplified and generalized masked diffusion for discrete data. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=xcqSOfHt4g.

Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *Proceedings of the 32nd International Conference on Machine Learning - Volume 37*, ICML'15, pp. 2256–2265. JMLR.org, 2015.

Yuerong Song, Xiaoran Liu, Ruixiao Li, Zhigeng Liu, Zengfeng Huang, Qipeng Guo, Ziwei He, and Xipeng Qiu. Sparse-dLLM: Accelerating diffusion llms with dynamic cache eviction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 33038–33046, 2026.

Yuxuan Song, Zheng Zhang, Cheng Luo, Pengyang Gao, Fan Xia, Hao Luo, Zheng Li, Yuehang Yang, Hongli Yu, Xingwei Qu, et al. Seed Diffusion: A large-scale diffusion language model with high-speed inference. *arXiv preprint arXiv:2508.02193*, 2025.

Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for LLM serving. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=meKEKDhdnx.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2023.127063. URL https://www.sciencedirect.com/science/article/pii/S0925231223011864.

Wenhao Sun, Rong-Cheng Tu, Yifu Ding, Zhao Jin, Jingyi Liao, Yongcheng Jing, and Dacheng Tao. Singular proxies for adaptive caching in diffusion language models. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=4Lucu401dx.

Jaesung Tae, Hamish Ivison, Sachin Kumar, and Arman Cohan. TESS 2: A large-scale generalist diffusion language model. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 21171–21188, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1029. URL https://aclanthology.org/2025.acl-long.1029/.

Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pp. 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

vLLM-Omni. Continuous batching for Step-Wise diffusion. https://docs.vllm.ai/projects/vllm-omni/en/v0.20.0/design/feature/diffusion_continuous_batching/, 2026. vLLM-Omni documentation.

Jinhao Wang, Zhexin Hu, Kangjie Zhou, Xin Zhou, and Fangfang Liu. GroupKV: Hierarchical kv cache management for long-context diffusion llm inference. *arXiv preprint arXiv:2609.17573*, 2026a.

Shihao Wang, Jiahao Chen, Yanqi Pan, Hao Huang, Yichen Hao, Xiangyu Zou, Wen Xia, Chongyang Qiu, Wentao Zhang, and PengFei Wang. ProphetKV: User-query-driven selective recomputation for efficient KV cache reuse in retrieval-augmented generation. In *Forty-third International Conference on Machine Learning*, 2026b. URL https://openreview.net/forum?id=SwGwmKIgUb.

Chengyue Wu, Hao Zhang, Shuchen Xue, Shizhe Diao, Yonggan Fu, Zhijian Liu, Pavlo Molchanov, Ping Luo, Song Han, and Enze Xie. Fast-dLLM v2: Efficient block-diffusion LLM. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=1NZ3DHF9nT.

Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. Fast-dLLM: Training-free acceleration of diffusion LLM by enabling KV cache and parallel decoding. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=3Z3Is6hnOT.

Junyi Wu, Tianchen Zhao, Shaoqiu Zhang, Linfeng Zhang, Guohao Dai, and Yu Wang. Elastic-dllm: Position preserving context compression and augmentation of diffusion llms. *arXiv preprint arXiv:2605.18165*, 2026c.

Tianyi Wu, Xiaoxi Sun, Yanhua Jiao, Yulin Li, Yixin Chen, Yun-Hao Cao, Yi-Qi Hu, and Zhuotao Tian. Dynamic-dLLM: Dynamic cache-budget and adaptive parallel decoding for training-free acceleration of diffusion LLM. In *The Fourteenth International Conference on Learning Representations*, 2026d. URL https://openreview.net/forum?id=SdnkB5pGbq.

xAI. grok-prompts: Prompts for the Grok chat assistant and the @grok bot on X. https://github.com/xai-org/grok-prompts, 2025. GitHub repository.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=NG7sS51zVF.

Zhongyu Xiao, Zhiwei Hao, Jianyuan Guo, Yong Luo, Jia Liu, Jie Xu, and Han Hu. Streaming-dLLM: Accelerating diffusion llms via suffix pruning and dynamic decoding. *arXiv preprint arXiv:2601.17917*, 2026.

Jing Xiong, Qi Han, Shansan Gong, Yunta Hsieh, Boyuan Zheng, Chengyue Wu, Chaofan Tao, Chenyang Zhao, and Ngai Wong. Prefilling-dLLM: Predictive prefilling for long-context inference in diffusion language models. *arXiv preprint arXiv:2606.10537*, 2026.

Haobo Xu, Sirui Chen, Yuanchen Bei, Lingjie Chen, Yuchen Yan, Dongqi Fu, Jingrui He, and Hanghang Tong. Predict, don't iterate: Efficient adaptive-length infilling for diffusion language models. *arXiv preprint arXiv:2609.02108*, 2026.

Jinnan Yang, Yan Wang, Zhen Bi, Kehao Wu, Xiaojie Li, Jungang Lou, Zechao Li, and Jing Liu. Wavefilter: Enhancing the long-context capability of diffusion llms via wavelet-guided kv cache filtering. *arXiv preprint arXiv:2606.00724*, 2026.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. CacheBlend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the twentieth European conference on computer systems*, pp. 94–109, 2025.

Yizhen Yao, Qinglin Zhu, Runcong Zhao, Xiangxiang Dai, Yanzheng Xiang, Yulan He, and Lin Gui. Follow the latent roadmap: Navigating revocable decoding for diffusion llms with anchor tokens. *arXiv preprint arXiv:2606.16847*, 2026.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7B: Diffusion large language models. *arXiv preprint arXiv:2508.15487*, 2025a.

Lu Ye, Ze Tao, Yong Huang, and Yang Li. ChunkAttention: Efficient self-attention with prefix-aware KV cache and two-phase partition. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11608–11620, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.623. URL https://aclanthology.org/2024.acl-long.623/.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Eighth Conference on Machine Learning and Systems*, 2025b. URL https://openreview.net/forum?id=RXPofAsL8F.

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)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Ying Yuan, Pengfei Zuo, Bo Wang, Zhangyu Chen, Zhipeng Tan, and Zhou Yu. DualMap: Enabling both cache affinity and load balancing for distributed LLM serving. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=zCadrJ32Xn.

Quqing Zhang, Kai Chen, Ning Liao, Zehao Lin, Bo Tang, Feiyu Xiong, Zhiyu Li, and Xiaoxing Wang. SparseX: Efficient segment-level kv cache sharing for interleaved llm serving. *arXiv preprint arXiv:2606.01751*, 2026.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2O: heavy-hitter oracle for efficient generative inference of large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NeurIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NeurIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

Fengqi Zhu, Zebin You, Yipeng Xing, Zenan Huang, Lin Liu, Yihong Zhuang, Guoshan Lu, Kangyu Wang, Xudong Wang, Lanning Wei, et al. Llada-moe: A sparse moe diffusion language model. *arXiv preprint arXiv:2509.24389*, 2025.

Fengqi Zhu, Rongzhen Wang, Shen Nie, Xiaolu Zhang, Chunwei Wu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. LLaDA 1.5: Variance-reduced preference optimization for large language diffusion models. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11425–11460, San Diego, California, United States, July 2026a. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.524. URL https://aclanthology.org/2026.acl-long.524/.

Zijian Zhu, Fei Ren, Zhanhong Tan, and Kaisheng Ma. ES-dLLM: Efficient inference for diffusion large language models by early-skipping. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=O2WvMkJbws.

Fengrui Zuo, Zhiwei Ke, Yiming Liu, Wenqi Lou, Chao Wang, and Xuehai Zhou. Window-diffusion: Accelerating diffusion language model inference with windowed token pruning and caching. *arXiv preprint arXiv:2601.20332*, 2026.

A Related Work

A.1 Development of Diffusion-Based Language Models

Diffusion-based language models adapt diffusion probabilistic models (Sohl-Dickstein et al., 2015; Ho et al., 2020) to text by iteratively denoising a corrupted sequence: the whole sequence is laid out from the first step, and predictions at masked positions are conditioned on global context rather than a prefix. This non-causal formulation appears in continuous-embedding models such as Diffusion-LM (Li et al., 2022), and discrete-state models such as D3PM (Austin et al., 2021a) and Multinomial Diffusion (Hoogeboom et al., 2021). Subsequent work such as SEDD (Lou et al., 2024), MDLM (Sahoo et al., 2024), MD4 (Shi et al., 2024), and RADD (Ou et al., 2025) refined this formulation with simpler objectives and stronger training, narrowing the gap to AR models.

A.2 DLLM Acceleration

ACache is orthogonal to most DLLM acceleration techniques. Many reduce intra-request cost through cache reuse and recomputation (Ma et al., 2025a; Liu et al., 2026c; Wu et al., 2026b; Hu et al., 2026; Ma et al., 2025b). Inference systems interleave recomputation and decoding across concurrent requests (Fan et al., 2025; Kedia et al., 2026), and offload KV state to host memory to retrieve a sparse subset on demand (Kwon et al., 2026; Wang et al., 2026a). Other work shrinks the context each step attends to, pruning distant suffix tokens or compressing the masked region (Zuo et al., 2026; Xiao et al., 2026; Chen et al., 2026; Wu et al., 2026c), and for long contexts sparsifies attention or filters what it caches (Long et al., 2026; Xiong et al., 2026; Yang et al., 2026). Some adjust the decoding schedule, varying block size or tokens committed per step (Kong et al., 2025; Lu et al., 2026; Luo et al., 2026a). Others decide at runtime which cached state merits reuse (Cheong et al., 2026; Sun et al., 2026; Wu et al., 2026d; Frumkin et al., 2026; He et al., 2026b; Lee et al., 2026a), or compute only the tokens that still change (Zhu et al., 2026b; Lee et al., 2026b). Several read that decision off attention (Huang et al., 2026; Song et al., 2026; Jiang et al., 2026; Nguyen-Tri et al., 2026; Liang et al., 2026; Luo et al., 2026b). ACache also uses attention, but to select request-specific Anchor Tokens, keeping the rest of the cache (Appendix D.6). ASRD uses the term for another object: generated tokens that stay stable across iterations, used to revoke earlier predictions (Yao et al., 2026). ACache targets cross-request reuse; BiCache shares that target but resolves reuse by depth rather than by token, and as originally designed covers prefixes alone (Go et al., 2026). ACache and these intra-request optimizations compose: it exposes reuse across requests, they cut the cost that remains.

A.3 Inference Abstractions

In AR LLM inference, prefix-cache reuse has shaped runtime abstractions at every layer of the inference stack: cache layout and prefix indexing (Kwon et al., 2023; Zheng et al., 2024), prefix-aware attention kernels (Ye et al., 2024; Pan et al., 2025; Ye et al., 2025b), eviction policies (Ouyang et al., 2026), request scheduling and cache affinity (Srivatsa et al., 2025; Yuan et al., 2026), cluster-level cache pooling (Qin et al., 2025), and modular reuse abstractions (Gim et al., 2024). CacheBlend studies non-prefix chunk reuse for AR RAG inference (Yao et al., 2025), and EPIC formalizes it as position-independent caching (Hu et al., 2025); both settle a reused span before generation begins, which a DLLM cannot do (Section 4). Subsequent AR systems push this direction further, sharing KV at segment granularity across requests (Zhang et al., 2026), reusing it per attention head (Liu et al., 2026b), spending the recomputation budget on the entries the query actually needs (Wang et al., 2026b), and tuning how much to recompute against hardware cost (Liu et al., 2026a). By treating shared context as an affix-level object whose consistency can be managed across requests, ACache points to an inference research direction for DLLMs analogous to prefix caching in AR LLMs, and one where bidirectional modeling may organize shared context more flexibly.

B System Design Details

B.1 Kernel Integration

To support the per-request read-slot map, the system implements a customized Triton (Tillet et al., 2019) attention kernel for ACache. In addition to the paged KV cache and ragged query/KV sequence

Algorithm 2 ACache attention kernel (one query tile, one head)

Input: query tile Q; `read_slot_map`; flat per-slot caches K, V; KV length kv_len
Output: attention output O for this tile

$m \leftarrow -\infty$, $\ell \leftarrow 0$, $acc \leftarrow 0$ ▷ online-softmax state
for $t \leftarrow 0$ to $kv_len - 1$ step $BLOCK_N$ **do**
  $slots \leftarrow$ `read_slot_map`$[t : t + BLOCK_N]$ ▷ physical slots for this KV tile
  $K_{\text{tile}} \leftarrow K[slots]$, $V_{\text{tile}} \leftarrow V[slots]$ ▷ direct slot gather, no block table
  $scores \leftarrow Q \cdot K_{\text{tile}}^{\top} \cdot sm_scale$
  $m_{\text{new}} \leftarrow \max(m, \text{rowmax}(scores))$
  $p \leftarrow \exp(scores - m_{\text{new}})$, $\alpha \leftarrow \exp(m - m_{\text{new}})$
  $\ell \leftarrow \alpha \cdot \ell + \text{rowsum}(p)$, $acc \leftarrow \alpha \cdot acc + p \cdot V_{\text{tile}}$
  $m \leftarrow m_{\text{new}}$
end for
$O \leftarrow acc/\ell$

Algorithm 3 Shared-affix rotation dispatch

Input: engine configuration max_num_seqs; a shared affix whose K was rotated for position zero but now starts at offset $shift \neq 0$
Output: correctly positioned K for the shared slots, as seen by attention

if $max_num_seqs \neq 1$ **then** ▷ configured for batching
  In Algorithm 2, score the shared-slot tiles with Q rotated by $-shift$ and the remaining tiles with Q ▷ one rotation, before the KV loop
else if a workspace can be reserved **then** ▷ configured for single-sequence inference
  $K' \leftarrow \text{rotate}(K, shift)$, $V' \leftarrow V$ ▷ V carries no position, so it is copied as is
  Write K', V' into the workspace
  Redirect `read_slot_map` to the workspace ▷ Algorithm 2 then runs unmodified
else ▷ no workspace available
  In Algorithm 2, rotate the shared keys by $shift$ as each tile is gathered ▷ stored cache untouched
end if

lengths, the kernel takes a flattened `read_slot_map`. For each KV tile, it loads the physical slots named by this map before gathering keys and values from the paged cache. Thus logical KV positions can point either to shared affix slots or to request-private slots, while the query tile remains a standard dense tile. Algorithm 2 details the per-tile loop, which keeps the tiled online-softmax accumulation of FlashAttention (Dao et al., 2022) and changes only how each tile is addressed: unlike a standard paged-attention kernel, which resolves each KV tile through a block table (`block_id`, `offset`) (Kwon et al., 2023), this kernel gathers keys and values directly by physical slot, so shared and private KV states are read through the same indirection.

FlashInfer already represents KV caches with block-sparse formats, including fine-grained blocks, and composes attention over shared-prefix and request-specific KV by merging their partial attention states through log-sum-exp (Juravsky et al., 2024; Ye et al., 2025b). Its shared-prefix use case does not need to address relocating a cached span across request-specific positions, since the shared prefix occupies the same initial positions in every request. ACache reuses one affix cache even when its reusable slots are interleaved with request-private Anchor slots and the affix begins at different offsets across requests; the read-slot map exposes these slots as one logical KV view, while the positional correction in Algorithm 3 lets the shared cache stay unchanged across offsets.

The runtime dispatches the shared-affix rotation correction (Section 4) based on the engine's configured maximum batch size, not on how many sequences a given batch actually holds. Algorithm 3 summarizes these cases. In the batched case, the runtime first groups a request's shared slots ahead of its private ones in the read-slot map—exact, since bidirectional attention imposes no order on the KV axis—so every KV tile is either all shared or all private and the counter-rotation applies once to the query tile rather than once per key slot. The other cases need no grouping: a materialized copy is already positioned for the request, and an affix starting at position zero takes no correction at all. A single-sequence request that cannot reserve the workspace needs no grouping either: the kernel rotates the shared keys inline as it reads them, leaving the stored cache untouched.

The ACache path preserves the same store-then-attend interface in both recomputation and block decoding. The runtime first writes newly produced KV states through the normal write-side slot mapping, then invokes the read-slot kernel over the effective KV view. The two phases differ only in

the query/write slice: recomputation uses the Anchor Tokens and request-specific positions, whereas decoding uses the current block. This keeps ACache's storage indirection inside the attention read path: neither phase materializes a blended copy of the shared affix cache.

B.2 Continuous Batching Across Caching Mechanisms

The inference loop of Algorithm 1 advances each request's currently active region (Section 4) without prescribing how the underlying intra-request caching mechanism defines it. In this codebase, both supported mechanisms instantiate it as a fixed-size block of `block_length` tokens and require a block to fully unmask before the sequence advances to the next one, but they differ in how they support continuous batching around this boundary. Under Fast-dLLM (Wu et al., 2026b), the runtime treats every completed block as a checkpoint: it advances each sequence whose block has just unmasked, retires those that have finished their last block, and refills the freed slots from the pending queue within the same pass, so newly admitted or newly advanced requests can join the very next decode step. Under dKV-Cache (Ma et al., 2025a), each step commits a fixed number of tokens rather than all those above a confidence threshold, so every sequence in a batch spends the same number of steps on each block. The runtime therefore admits a batch, runs it through all of its blocks, and deallocates it as a whole before admitting the next: requests join and leave only at batch boundaries.

C Evaluation Details

C.1 Experimental Setup

Quality evaluations use the Language Model Evaluation Harness (`lm-eval`) (Gao et al., 2024) with model-specific wrappers for LLaDA and Dream. On this backend, affix keys are stored before rotary rotation and rotated to the affix's request-time position when read, rather than stored for position zero and corrected on the query side as in the runtime (Section 4). Both score the affix at its absolute request-time positions, so placement is exact on either backend. We measure accuracy on this backend rather than on the runtime to reduce variance: under continuous batching, a request's output in the runtime can shift with the other requests that share its batch. At batch sizes 1, 4, and 16, runtime accuracy at Anchor ratio 0.2 stays within 2.0 points of the Transformers backend's in all 24 GSM8K and MBPP settings of Figure 5, and is 0.1 points higher on average. GSM8K and MBPP use standard task data with the affix serialization in Appendix C.2. For BABILong, we add a custom task YAML that loads the `0k/qa1` split of `RMT-team/babilong-1k-samples` and scores it with exact match.

C.2 Prompt Construction

All prompts are serialized with the corresponding model chat template; the templates below show the message contents before tokenizer-specific role and control tokens are added. Let Ex denote the serialized few-shot examples used as the shared affix, Qry the task query, Ans the final prompt used in the infix layout to direct the model back to the initial query, and Mask the masked generation region. The prefix layout is Ex Qry Mask. The infix layout is $\mathsf{Qry}_{\mathrm{head}}$ Ex Ans Mask, where examples are inserted between the query header and this final prompt. The suffix layout is Qry Mask Ex.

GSM8K. For GSM8K, the query message is:

```
Question: {question}
Let's think step by step.
Answer:
```

Few-shot answers are normalized to:

```
{reasoning}
The answer is {final_answer}
```

For infix placement, the query header is:

```
Question: {question}

Example(s):
```

and the final answer prompt is:

```
The examples above are demonstrations only.
Let's come back to the question at the beginning and think step by step.
Answer:
```

MBPP. For MBPP, the query message is:

```
You are an expert Python programmer, and here is your task: {question}
Your code should pass these tests:

{test_1}
{test_2}
{test_3}
[BEGIN]
```

Few-shot answers are normalized to:

```
{code}
[DONE]
```

For infix placement, the query header replaces the final `[BEGIN]` marker with a blank line followed by:

```
Example(s):
```

and the final answer prompt is:

```
The examples above demonstrate the expected response pattern.
Let's come back to the task at the beginning and write the Python code in
that pattern.
Do not include any explanation, comments outside the code, markdown fences,
or test cases.
[BEGIN]
```

BABILong-0k/qa1. For BABILong, the query message is:

```
Story:
{story}
Question: {question}
Answer with exactly one location name.
Location:
```

Few-shot answers are the stripped target location:

```
{target}
```

For infix placement, the query header is:

```
Story:
{story}
Question: {question}

Example(s):
```

and the final answer prompt is:

```
End of examples.
Let's come back to the story and question at the beginning.
Answer with exactly one location name.
Location:
```

Few-shot affix serialization. Few-shot examples are sampled from the `main`/`train` split of `gsm8k` for GSM8K and the `full`/`prompt` split of `google-research-datasets/mbpp` for MBPP; BABILong

has no separate example pool, so its examples come from the same `0k/qa1` split that is evaluated. In prefix and suffix modes, each sampled few-shot example is serialized as a user query followed by its normalized assistant answer. In infix mode, the shared affix is a plaintext block:

```
Example {i}:
{query message}
{normalized answer}
```

with examples separated by blank lines. The query header, example block, and final answer prompt are joined with blank lines within a single user message.

C.3 Baseline Criterion Adaptations

Table 5 matches CacheBlend, ProphetKV, and BiCache to ACache's Anchor ratio ρ, so all four criteria spend the same recomputation budget. Each baseline keeps ACache's generation loop and cache layout unchanged, changing only what it recomputes.

CacheBlend and ProphetKV both select at the token level, like ACache: each scores the L affix tokens and recomputes the top-$K = \lceil \rho L \rceil$ by that score, differing only in the scoring signal. CacheBlend (Yao et al., 2025) scores each affix token by high KV deviation: the ℓ_2 distance between its key and value entries in the affix-only cache and in the full-prompt cache, summed across layers, without using which masked generation positions attend to it. We obtain that full-prompt cache from an actual prefill at every layer, whereas CacheBlend prefills only the first layer and narrows the token set layer by layer, so this criterion is given information the original cannot obtain. ProphetKV (Wang et al., 2026b) instead scores each affix token by the attention that the request-owned prompt places on it, rather than the masked generation positions ACache uses, averaged across layers, heads, and query positions. Its original criterion scores by attention from the user query, which its retrieval-augmented setting places at the end of the prompt; since a DLLM affix can sit before, inside, or after the request-owned tokens, we anchor the signal to them wherever they fall.

BiCache (Go et al., 2026) selects on a different axis: it decides which *layers*, not which tokens, keep a stale cache. We match it to the same ratio ρ by recomputing the affix jointly with the request in the last $\lceil \rho N \rceil$ of the model's N layers, while the remaining, shallower layers reuse the shared affix cache untouched: request tokens alone traverse the shallow layers, and the affix rejoins the sequence only at the boundary layer. This adapts BiCache's layer-selection idea to ACache's ratio-based budget, without BiCache's own offline shared-prefix-ratio lookup or its periodic recompute interval. Where Table 5 and Table 6 instead report BiCache's own implementation, we run the released code at commit `82a8d0122807daf58ceacd6a121201b8355f46c1` of `https://github.com/OSSS-KU/BiCache`, changing only its fixed per-step decoding schedule to the 0.9 confidence threshold of Section 5. The release supports LLaDA alone; we port it to Dream, reprofiling its layer policy there.

D Additional Evaluation

D.1 Anchor Stability

Figure 3 illustrates on three samples that importance concentrates on a small subset of affix tokens and that this subset stays stable as decoding proceeds. Table 7 measures both over the full test sets of GSM8K (1,319 questions) and MBPP (500), with 1- and 2-shot affixes at every position, on Fast-dLLM at Anchor ratio 0.2. We omit BABILong, whose 8-token answer fits one block and so leaves no drift to measure. Decoding runs with full recomputation, so ACache does not shape the trajectory. At each of the 8 block boundaries b, Fast-dLLM already runs a full-sequence forward pass; from it we compute each affix token's importance s_j (Section 3) over the positions still masked, and take the top $K = \lceil \rho L \rceil$ tokens as the current top-K set. We compare that set with two reference sets and, as a baseline, a random set of the same size: Fresh, the top-K set at $b = 1$, which isolates drift during decoding; and ACache's Anchors, selected against the precomputed affix cache (Section 3). Jaccard similarity is the fraction of positions two sets have in common (intersection over union). Mass recall divides the importance a reference set captures at b by that of the current top-K set, so swapping tokens of near-equal importance barely lowers it. One MBPP question (infix, 2-shot, LLaDA) is excluded because its attention scores at some boundary were not finite.

Table 7: **Anchor stability at Anchor ratio 0.2.** Fast-dLLM, full recomputation; all values in %, means over each test set. At each block boundary, a set is compared with the current top-K set: Random, Fresh (the first's), or ACache's Anchors (averaged). Share: the first top-K set's importance.

Dataset	Affix	Shots	Model	Share	Jaccard similarity				Mass recall		
					Random	Fresh		ACache	Fresh		ACache
						2nd	Last		2nd	Last	
GSM8K	Prefix	1	LLaDA	62.7	11.3	71.6	56.8	38.9	98.5	95.8	76.7
			Dream	71.8	11.3	78.8	57.4	54.4	97.8	89.5	89.3
		2	LLaDA	62.6	11.2	73.4	52.5	35.2	98.5	95.1	91.4
			Dream	73.4	11.2	84.9	58.2	62.8	99.4	93.6	94.3
	Infix	1	LLaDA	51.3	11.3	48.8	54.9	33.9	91.2	92.4	84.7
			Dream	62.2	11.5	68.7	69.0	64.4	94.2	95.5	94.6
		2	LLaDA	60.2	11.1	58.6	53.4	29.1	95.8	95.4	88.6
			Dream	66.1	11.2	71.0	62.9	58.6	96.5	94.3	93.5
	Suffix	1	LLaDA	67.4	11.3	70.8	55.5	45.8	98.4	96.3	94.6
			Dream	59.8	11.3	84.5	61.4	57.1	98.3	91.6	86.8
		2	LLaDA	66.8	11.2	81.9	63.1	50.1	99.1	97.3	93.7
			Dream	76.7	11.2	89.4	68.8	69.7	99.3	96.8	91.6
MBPP	Prefix	1	LLaDA	59.0	11.4	70.2	68.8	63.1	97.9	97.6	95.4
			Dream	81.2	11.5	79.0	65.5	69.9	99.5	98.5	98.7
		2	LLaDA	58.2	11.2	68.0	62.9	57.7	97.4	96.2	95.7
			Dream	81.4	11.2	75.7	64.1	68.5	99.3	98.3	98.8
	Infix	1	LLaDA	47.2	11.2	62.1	58.0	55.9	95.2	94.9	92.2
			Dream	48.6	11.3	69.6	54.4	68.9	95.8	90.9	96.3
		2	LLaDA	56.2	11.2	62.6	55.5	51.7	95.8	95.3	93.2
			Dream	56.3	11.2	71.3	62.6	66.8	94.6	93.1	93.4
	Suffix	1	LLaDA	63.9	11.4	76.7	65.3	56.6	99.1	96.7	95.0
			Dream	67.3	11.4	73.1	64.7	63.8	97.9	95.9	95.5
		2	LLaDA	64.1	11.2	76.9	66.8	59.8	98.7	97.6	96.0
			Dream	69.1	11.2	77.9	64.5	65.5	98.1	96.7	95.1

Importance concentrates: the top 20% of affix tokens receive 47–81% of the mask-to-affix attention, against 20% under uniform importance. The Fresh set's membership drifts: by the last boundary its Jaccard similarity is 52–69%, still well above the 11–12% of random sets. Its importance drifts far less: mass recall is at least 91% at the second boundary and 89% at the last, whereas a random set reaches about ρ/Share, or 25–42%. A fresh selection at request start thus captures nearly as much importance as one made later, so one selection per request suffices. ACache's Anchors share fewer positions with the current top-K set, since they are scored against the stale precomputed affix cache: their Jaccard similarity is 29–70%, at least 2.6 times a random set's. Their mass recall averages 77–99% and falls below 84% only for LLaDA on 1-shot GSM8K prefix, where accuracy stays within 1.9 points of full recomputation (Figure 5). Scoring with fresh affix KV would require recomputing the whole affix, the cost ACache avoids. At ratio 0.3, on the same runs, the same holds: the top 30% capture 58–86%, Fresh mass recall is at least 93%, and the Anchors' average mass recall is 79–99%.

D.2 Anchor Selection Overhead

Table 8 separates ACache's one-time Anchor selection cost into preparation, the mask-to-affix attention forward pass, and top-K selection. Selection costs 44.9–72.4 ms per request. The forward pass accounts for 98.5–98.9% of it, while preparation costs at most 0.60 ms and top-K at most 0.21 ms. The cost tracks the affix length rather than the decoding loop: it grows from 44.9–55.4 ms at 1-shot to 50.7–72.4 ms at 4-shot, as the shared span grows from 172–213 to 951–1016 tokens. The selector is thus a single extra forward pass per request, forming queries only for request-owned tokens, never for the L affix positions (Section 3).

We report batch size 1 because Anchor selection runs one request at a time, so the inference batch size does not change what the selector does. Measurement agrees: holding everything else fixed, per-request selection cost varies by at most 1.8% across batch sizes 1, 4, and 16, except for Dream on 1-shot GSM8K infix, where it varies by 4.3%. Affix position matters more. Relative to prefix, suffix reuse costs 0.8–4.7% more and infix reuse 2.1–15.6% more. Preparation, which builds the

Table 8: **Anchor selection cost per request (ms) under prefix reuse, batch size 1.** Profiled on the runs of Table 2 (LLaDA) and Table 9 (Dream). Affix reports the shared span's length in tokens.

Model	Dataset	Shot	Affix	Preparation	Forward	Top-K	Total	Forward share
LLaDA	GSM8K	1	213	0.49	50.05	0.16	50.69	98.7%
		2	549	0.52	56.46	0.16	57.14	98.8%
		4	981	0.57	66.02	0.18	66.76	98.9%
	MBPP	1	178	0.55	54.70	0.16	55.40	98.7%
		2	455	0.55	61.40	0.19	62.14	98.8%
		4	1016	0.60	71.57	0.21	72.38	98.9%
Dream	GSM8K	1	213	0.49	44.26	0.16	44.90	98.6%
		2	538	0.51	46.42	0.16	47.09	98.6%
		4	957	0.55	49.95	0.18	50.68	98.6%
	MBPP	1	172	0.52	47.51	0.18	48.21	98.5%
		2	434	0.53	49.27	0.20	50.01	98.5%
		4	951	0.59	53.91	0.20	54.70	98.5%

Table 9: **Dream recompute latency (ms) under prefix reuse.** LLaDA: Table 2.

Dataset	Batch	1-shot		2-shot		4-shot	
		Fast-dLLM	ACache	Fast-dLLM	ACache	Fast-dLLM	ACache
GSM8K	1	52.8	41.6 (21.3% ↓)	72.5	46.4 (35.9% ↓)	111.7	54.8 (50.9% ↓)
	4	60.7	46.6 (23.3% ↓)	87.0	52.3 (39.9% ↓)	132.4	63.5 (52.0% ↓)
	16	96.5	70.2 (27.3% ↓)	147.0	80.4 (45.3% ↓)	233.5	102.3 (56.2% ↓)
MBPP	1	54.4	45.2 (17.0% ↓)	72.7	49.2 (32.4% ↓)	119.0	60.3 (49.4% ↓)
	4	70.9	56.0 (21.0% ↓)	96.7	62.2 (35.7% ↓)	162.5	76.7 (52.8% ↓)
	16	130.2	98.6 (24.3% ↓)	184.3	110.2 (40.2% ↓)	310.6	142.8 (54.0% ↓)

read-slot map for a relocated span, rises from about 0.5 ms to 1.4–2.7 ms and covers the smaller gaps; the widest infix gaps come from the forward pass, because the infix layout adds a query header and answer prompt to the tokens the pass forms queries for (Appendix C.2). Precomputing the shared affix cache is a separate cost, paid once per shared span rather than per request: 0.76–1.24 s, amortized across the requests that reuse it. A $4.5\times$ longer affix takes only $1.1\times$ longer to precompute, so most of that time is fixed setup rather than the forward pass itself.

Both the selector's cost and the recomputation it saves scale with the workload. Anchor selection is 2.0–4.8% of ACache's per-request end-to-end time at batch size 1, rising to 7.5–11.1% at batch size 16, where the decode work attributable to each request shrinks while the selector's one pass does not. The recompute savings grow faster: across both models, ACache reduces recompute latency by 17.0–56.7% and improves throughput in every tested batch-4 and batch-16 setting (Table 3). At batch size 1, where the shortest affix leaves the least to save, selection is repaid in seven of the eight settings with more than one shot (1.04–$1.26\times$); the exception, Dream on GSM8K at 2-shot, breaks even. One caveat applies to all of these measurements: the selection pass runs with module compilation disabled, since its attention hook has to observe uncompiled forwards, while ACache's decode and recomputation passes run compiled. The reported selector cost is therefore an upper bound.

D.3 Dream System Efficiency

Table 9 complements the LLaDA recomputation results in Table 2; Table 3 already reports throughput for both models. ACache reduces Dream recompute latency in every tested setting, by 17.0% to 56.2%. These gains are largest with the longest shared affix: at 4-shot, the reduction is 49.4–56.2% across both datasets and all batch sizes, agreeing with LLaDA's to within 2.5 points. At 1- and 2-shot, Dream's reductions are lower than LLaDA's in 11 of the 12 settings, by up to 13.6 points.

Table 10 complements the LLaDA memory results in Table 4. Absolute KV cache memory is lower for Dream because it has 4 grouped-query KV heads in each of 28 layers where LLaDA has 32 in each of 32, about $9\times$ smaller per token; the meaningful comparison is therefore the within-model Fast-dLLM-to-ACache reduction. Those reductions are identical to LLaDA's in 16 of the 18 settings, differing only on MBPP at batch size 16, where the baseline occupies a different number of

Table 10: **Dream peak KV cache memory usage (GB) under prefix reuse.** LLaDA: Table 4.

Dataset	Batch	1-shot		2-shot		4-shot	
		Fast-dLLM	ACache	Fast-dLLM	ACache	Fast-dLLM	ACache
GSM8K	1	0.04	0.04 (0.0%)	0.05	0.08 (50.0% ↑)	0.08	0.10 (16.7% ↑)
	4	0.16	0.12 (25.0% ↓)	0.22	0.16 (25.0% ↓)	0.33	0.22 (33.3% ↓)
	16	0.66	0.45 (31.2% ↓)	0.88	0.51 (42.2% ↓)	1.31	0.71 (45.8% ↓)
MBPP	1	0.23	0.23 (0.0%)	0.25	0.25 (0.0%)	0.27	0.29 (5.0% ↑)
	4	0.36	0.33 (7.7% ↓)	0.41	0.36 (13.3% ↓)	0.52	0.41 (21.1% ↓)
	16	0.83	0.68 (18.0% ↓)	1.07	0.71 (33.3% ↓)	1.50	0.90 (40.0% ↓)

Table 11: **dKV-Cache periodic full-cache recompute latency (ms) under prefix reuse.** OOM marks a run that exceeded GPU memory.

Model	Dataset	Batch	1-shot		2-shot		4-shot	
			dKV-Cache	ACache	dKV-Cache	ACache	dKV-Cache	ACache
LLaDA	GSM8K	1	54.1	38.1 (29.6% ↓)	88.4	44.9 (49.2% ↓)	120.4	58.3 (51.6% ↓)
		4	166.2	117.4 (29.4% ↓)	285.7	152.1 (46.8% ↓)	439.5	181.9 (58.6% ↓)
		16	627.4	435.2 (30.6% ↓)	1053.7	533.4 (49.4% ↓)	1674.0	672.7 (59.8% ↓)
	MBPP	1	55.8	43.1 (22.8% ↓)	88.2	48.8 (44.6% ↓)	129.2	63.9 (50.6% ↓)
		4	175.3	138.5 (21.0% ↓)	263.9	162.3 (38.5% ↓)	477.1	205.1 (57.0% ↓)
		16	647.5	484.8 (25.1% ↓)	1019.6	586.9 (42.4% ↓)	1997.7	784.2 (60.7% ↓)
Dream	GSM8K	1	52.8	41.6 (21.3% ↓)	72.4	46.4 (35.9% ↓)	111.5	54.6 (51.0% ↓)
		4	168.3	121.0 (28.1% ↓)	270.9	139.4 (48.5% ↓)	423.5	177.6 (58.1% ↓)
		16	622.7	434.6 (30.2% ↓)	1021.0	523.9 (48.7% ↓)	1609.1	654.6 (59.3% ↓)
	MBPP	1	54.1	45.0 (16.9% ↓)	72.1	49.1 (31.9% ↓)	119.6	60.4 (49.5% ↓)
		4	178.4	136.3 (23.6% ↓)	259.3	153.5 (40.8% ↓)	439.2	196.8 (55.2% ↓)
		16	642.5	483.0 (24.8% ↓)	956.2	560.5 (41.4% ↓)	*OOM*	791.7

cache blocks. The batch-size-1 increases come from the same block-granularity effect described in Section 5.2, and the largest increase is 0.03 GB, against savings that reach 0.60 GB at batch size 16.

D.4 dKV-Cache System Efficiency

Tables 11–13 repeat the recompute latency, throughput, and peak KV cache measurements of Tables 2–4 with ACache built on dKV-Cache, restricted to the prefix affix position. dKV-Cache recomputes its full cache every 8 decoding steps. In every setting that both mechanisms completed, ACache reduces dKV-Cache recompute latency by 16.9% to 60.7%, and unlike on Fast-dLLM it improves throughput in all of them, by $1.03\times$ to $1.60\times$, including at batch size 1 with a 1-shot affix. dKV-Cache runs $3.2\times$ to $7.4\times$ slower than Fast-dLLM on the same setting, so absolute rates here are far lower than in Table 3. That gap is also why batch size 1 gains here where it did not there: Anchor selection is one forward pass per request rather than per step (Appendix D.2), so its share of a longer run is smaller. Peak KV cache usage drops by up to 46.3%, following the same batch-size-1 allocation effect as Table 4 and closely tracking its LLaDA values, since peak usage is set by the paged memory manager both mechanisms share. In the one remaining setting, Dream on MBPP at 4-shot and batch 16, standalone dKV-Cache exceeded GPU memory while ACache completed the run, so all three tables report ACache alone there. The failure is one of activation memory in the recomputation pass rather than of cache size; ACache recomputes fewer positions and so needs less of it.

D.5 Infix and Suffix System Efficiency

Tables 14–19 repeat the recompute latency, throughput, and peak KV cache measurements of Tables 2–4 with the shared affix placed as an infix or a suffix. Recompute latency reductions remain similar across positions: 13.5–54.5% for infix and 15.8–56.0% for suffix, close to the 17.0–56.7% range for prefix. Throughput follows the same pattern as prefix, reaching up to $1.54\times$ for infix and $1.66\times$ for suffix against $1.71\times$ for prefix, but the shortest, 1-shot affix takes longer to amortize: ACache is 1–13% slower than baseline at batch sizes 1 and 4 for both infix and suffix, and 2% slower in one setting at batch size 16, whereas every prefix batch-4 and batch-16 setting already improves. Beyond 1-shot, only Dream on GSM8K infix at 2-shot and batch size 1 is slower, by 1%. Peak KV cache

Table 12: **End-to-end throughput under prefix reuse on dKV-Cache.** Measured in generated tokens per second. OOM marks a run that exceeded GPU memory.

Model	Dataset	Batch	1-shot		2-shot		4-shot	
			dKV-Cache	ACache	dKV-Cache	ACache	dKV-Cache	ACache
LLaDA	GSM8K	1	33	35 (1.08×)	28	33 (1.21×)	24	31 (1.30×)
		4	56	62 (1.11×)	43	56 (1.29×)	33	50 (1.51×)
		16	68	77 (1.14×)	51	69 (1.35×)	38	60 (1.60×)
	MBPP	1	32	34 (1.05×)	28	33 (1.18×)	23	30 (1.29×)
		4	55	59 (1.08×)	45	55 (1.22×)	32	48 (1.51×)
		16	66	73 (1.11×)	52	67 (1.27×)	35	56 (1.59×)
Dream	GSM8K	1	32	33 (1.03×)	29	32 (1.10×)	25	31 (1.21×)
		4	59	64 (1.09×)	49	61 (1.24×)	39	55 (1.41×)
		16	72	80 (1.11×)	56	72 (1.28×)	44	65 (1.48×)
	MBPP	1	32	33 (1.03×)	30	32 (1.08×)	25	30 (1.21×)
		4	58	63 (1.07×)	50	59 (1.19×)	37	52 (1.40×)
		16	71	77 (1.08×)	59	71 (1.22×)	*OOM*	56

Table 13: **Peak KV cache memory usage (GB) under prefix reuse on dKV-Cache.** OOM marks a run that exceeded GPU memory.

Model	Dataset	Batch	1-shot		2-shot		4-shot	
			dKV-Cache	ACache	dKV-Cache	ACache	dKV-Cache	ACache
LLaDA	GSM8K	1	0.38	0.38 (0.0%)	0.50	0.75 (50.0% ↑)	0.75	0.88 (16.7% ↑)
		4	1.50	1.12 (25.0% ↓)	2.00	1.50 (25.0% ↓)	3.00	2.00 (33.3% ↓)
		16	6.00	4.12 (31.2% ↓)	8.00	4.50 (43.8% ↓)	12.00	6.50 (45.8% ↓)
	MBPP	1	2.12	2.12 (0.0%)	2.25	2.25 (0.0%)	2.50	2.62 (5.0% ↑)
		4	3.25	2.88 (11.5% ↓)	3.75	3.00 (20.0% ↓)	4.75	3.75 (21.1% ↓)
		16	7.50	6.12 (18.3% ↓)	9.75	6.38 (34.6% ↓)	13.75	8.25 (40.0% ↓)
Dream	GSM8K	1	0.04	0.04 (0.0%)	0.05	0.08 (50.0% ↑)	0.08	0.10 (16.7% ↑)
		4	0.16	0.12 (25.0% ↓)	0.22	0.16 (25.0% ↓)	0.33	0.22 (33.3% ↓)
		16	0.66	0.45 (31.2% ↓)	0.88	0.49 (43.7% ↓)	1.30	0.70 (46.3% ↓)
	MBPP	1	0.23	0.23 (0.0%)	0.25	0.25 (0.0%)	0.27	0.29 (5.0% ↑)
		4	0.34	0.31 (8.0% ↓)	0.41	0.33 (20.0% ↓)	0.52	0.41 (21.1% ↓)
		16	0.79	0.67 (15.5% ↓)	1.04	0.70 (32.9% ↓)	*OOM*	0.90

usage drops by up to 41.7%, vs. 45.8% for prefix; at batch size 1 it rises by up to 0.63 GB, vs. 0.25 GB for prefix, as a relocated affix also occupies a workspace (Algorithm 3).

D.6 Non-Anchor Cache Ablation

Table 20 compares ACache with a variant that selects the same Anchor Tokens but discards every non-anchor affix cache entry. This tests whether ACache reduces to token-level KV compression or eviction (Zhang et al., 2023; Xiao et al., 2024; Li et al., 2024; Pang et al., 2024; Huang et al., 2026; Song et al., 2026), which keeps only important tokens: if the non-anchor tokens were disposable, discarding them should cost nothing once the Anchors are recomputed. The two variants coincide at Anchor ratio 1.0, where nothing is left to discard; the table uses the two ratios of Table 5. Discarding costs LLaDA 8.0 points of average accuracy at Anchor ratio 0.2 and 7.2 at 0.3, concentrated in MBPP. On Dream the averages differ by at most 1.2 points, with the variant ahead at 0.2 and behind at 0.3, and it leads in several individual settings, by up to 13.7 points on MBPP suffix at 0.2. The non-anchor affix cache is thus load-bearing on LLaDA but not on Dream. Keeping it is the safe default, and cheap in memory: the read-slot map stores a single copy shared by every request (Section 4).

D.7 Top, Random, and Bottom Selection

Table 21 keeps ACache's recomputation budget fixed and changes only which affix tokens are recomputed: those ACache ranks most important (Top, its default), a uniformly random subset of the same size drawn per request (Random), or those it ranks least important (Bottom). Accuracy follows the ranking. Outside BABILong suffix, Top matches or beats both alternatives in every setting at both ratios, and Random stays at or above Bottom in all but three, trailing it there by at most 0.3

Table 14: **Recompute latency (ms) under infix reuse.** The shared affix sits between the query header and answer prompt rather than before the query.

Model	Dataset	Batch	1-shot		2-shot		4-shot	
			Fast-dLLM	ACache	Fast-dLLM	ACache	Fast-dLLM	ACache
LLaDA	GSM8K	1	56.6	42.7 (24.5% ↓)	91.6	47.3 (48.4% ↓)	123.6	61.9 (49.9% ↓)
		4	62.9	50.5 (19.7% ↓)	101.7	55.5 (45.5% ↓)	140.7	69.6 (50.6% ↓)
		16	90.7	72.0 (20.7% ↓)	148.7	80.3 (46.0% ↓)	215.0	97.9 (54.5% ↓)
	MBPP	1	61.1	49.1 (19.6% ↓)	94.2	55.8 (40.7% ↓)	138.0	69.7 (49.4% ↓)
		4	67.6	56.0 (17.3% ↓)	106.3	63.5 (40.3% ↓)	152.6	77.1 (49.5% ↓)
		16	86.0	73.3 (14.7% ↓)	133.6	84.4 (36.9% ↓)	211.5	104.1 (50.8% ↓)
Dream	GSM8K	1	53.5	43.5 (18.7% ↓)	74.2	47.8 (35.6% ↓)	111.9	57.1 (48.9% ↓)
		4	60.4	49.9 (17.4% ↓)	88.1	56.1 (36.3% ↓)	135.0	66.4 (50.8% ↓)
		16	94.5	76.2 (19.4% ↓)	159.3	88.2 (44.6% ↓)	213.6	104.6 (51.0% ↓)
	MBPP	1	57.4	48.3 (15.9% ↓)	74.4	52.7 (29.3% ↓)	121.3	63.5 (47.7% ↓)
		4	73.0	63.1 (13.5% ↓)	103.4	73.6 (28.9% ↓)	171.9	84.9 (50.6% ↓)
		16	135.5	110.2 (18.7% ↓)	200.7	129.0 (35.7% ↓)	326.0	159.1 (51.2% ↓)

Table 15: **End-to-end throughput under infix reuse.** Measured in generated tokens per second.

Model	Dataset	Batch	1-shot		2-shot		4-shot	
			Fast-dLLM	ACache	Fast-dLLM	ACache	Fast-dLLM	ACache
LLaDA	GSM8K	1	105	98 (0.93×)	92	95 (1.03×)	82	88 (1.08×)
		4	262	250 (0.95×)	202	234 (1.16×)	162	207 (1.28×)
		16	388	392 (1.01×)	273	349 (1.28×)	193	297 (1.54×)
	MBPP	1	198	186 (0.94×)	162	178 (1.10×)	119	146 (1.22×)
		4	368	349 (0.95×)	265	326 (1.23×)	183	266 (1.45×)
		16	445	445 (1.00×)	313	408 (1.30×)	205	300 (1.46×)
Dream	GSM8K	1	108	108 (0.99×)	88	88 (0.99×)	58	62 (1.07×)
		4	247	243 (0.98×)	150	160 (1.06×)	81	91 (1.13×)
		16	325	332 (1.02×)	174	191 (1.09×)	87	102 (1.17×)
	MBPP	1	160	152 (0.95×)	124	124 (1.00×)	65	70 (1.08×)
		4	274	265 (0.97×)	175	185 (1.05×)	77	87 (1.13×)
		16	328	333 (1.02×)	191	206 (1.08×)	81	93 (1.15×)

points. On average, Top leads Random by 4.1 points at both ratios, and Random leads Bottom by 3.2 at ratio 0.2 and 4.9 at 0.3. The gaps concentrate on MBPP infix and suffix and on BABILong infix: at ratio 0.2, MBPP suffix drops from 25.1% under Top to 18.6% under Random and 8.5% under Bottom. ACache's ranking thus tracks how much recomputing each affix token helps. BABILong suffix, where recomputation lowers accuracy (Section 5.1), reverses the order.

D.8 System-Prompt Reuse

Following BiCache (Go et al., 2026), we reuse a system prompt as the shared span: exactly the GSM8K system prompt in its official implementation, which builds on the role and safety instructions of xAI's Grok 4 system prompt (xAI, 2025) and asks for a final line "#### number". As there, the user message places the question after "Question:" and ends with "Answer:" on a new line. The serialized system message is the shared prefix, 1,334 tokens under LLaDA's chat template and 1,270 under Dream's. We measure accuracy on the runtime itself over all 1,319 test questions, scored with lm-eval, at batch size 1 so that no other request shares a batch (Appendix C.1); on LLaDA, Anchor ratio 1 reproduces Fast-dLLM's responses on all 1,319 questions. BiCache runs as in Appendix C.3.

Table 22 shows what direct reuse loses. On LLaDA, it scores 19.8% against 69.0% under full recomputation, and only 25% of its responses contain the requested "####" line, against 89%: the stale cache fails to convey that instruction. Anchors restore most of the loss, reaching 59.1% at ratio 0.2 and 65.4% at 0.3; BiCache, at a mean equivalent ratio of 0.50, recomputes more and reaches 66.9%, 1.6 points above ratio 0.3. On Dream, the stale cache costs little: every setting but BiCache stays within 1.0 point of full recomputation. BiCache instead loses accuracy: at ratio 0.57 it scores 70.0%, 9.4 points below full recomputation and 8.5 below even direct reuse.

Table 16: **Peak KV cache memory usage (GB) under infix reuse.**

Model	Dataset	Batch	1-shot		2-shot		4-shot	
			Fast-dLLM	ACache	Fast-dLLM	ACache	Fast-dLLM	ACache
LLaDA	GSM8K	1	0.38	0.62 (66.7% ↑)	0.62	1.12 (80.0% ↑)	0.75	1.38 (83.3% ↑)
		4	1.50	1.38 (8.3% ↓)	2.12	2.00 (5.9% ↓)	3.00	2.50 (16.7% ↓)
		16	6.00	4.38 (27.1% ↓)	8.12	5.50 (32.3% ↓)	12.00	7.00 (41.7% ↓)
	MBPP	1	2.12	2.25 (5.9% ↑)	2.25	2.62 (16.7% ↑)	2.50	3.12 (25.0% ↑)
		4	3.25	3.25 (0.0%)	3.75	3.75 (0.0%)	4.88	4.25 (12.8% ↓)
		16	7.75	6.75 (12.9% ↓)	9.88	7.88 (20.3% ↓)	14.00	8.75 (37.5% ↓)
Dream	GSM8K	1	0.04	0.07 (66.7% ↑)	0.05	0.12 (125.0% ↑)	0.08	0.15 (83.3% ↑)
		4	0.16	0.15 (8.3% ↓)	0.22	0.23 (6.3% ↑)	0.33	0.27 (16.7% ↓)
		16	0.66	0.48 (27.1% ↓)	0.88	0.60 (31.3% ↓)	1.31	0.77 (41.7% ↓)
	MBPP	1	0.23	0.25 (5.9% ↑)	0.25	0.29 (16.7% ↑)	0.27	0.34 (25.0% ↑)
		4	0.36	0.34 (3.8% ↓)	0.41	0.41 (0.0%)	0.52	0.46 (10.5% ↓)
		16	0.85	0.72 (14.5% ↓)	1.07	0.83 (21.8% ↓)	1.50	0.96 (36.4% ↓)

Table 17: **Recompute latency (ms) under suffix reuse.** The shared affix sits after the masked generation region rather than before the query.

Model	Dataset	Batch	1-shot		2-shot		4-shot	
			Fast-dLLM	ACache	Fast-dLLM	ACache	Fast-dLLM	ACache
LLaDA	GSM8K	1	56.4	39.7 (29.6% ↓)	91.3	46.4 (49.2% ↓)	123.6	61.1 (50.6% ↓)
		4	62.5	46.0 (26.5% ↓)	101.7	54.1 (46.8% ↓)	141.9	68.9 (51.5% ↓)
		16	90.6	66.6 (26.5% ↓)	150.4	80.4 (46.6% ↓)	220.0	96.8 (56.0% ↓)
	MBPP	1	59.3	45.4 (23.4% ↓)	91.7	50.8 (44.6% ↓)	134.4	66.2 (50.7% ↓)
		4	63.3	51.6 (18.5% ↓)	97.8	56.8 (41.9% ↓)	148.4	73.5 (50.4% ↓)
		16	81.0	68.2 (15.8% ↓)	122.4	72.5 (40.8% ↓)	199.8	94.4 (52.7% ↓)
Dream	GSM8K	1	52.8	42.1 (20.2% ↓)	72.5	46.8 (35.4% ↓)	112.5	56.1 (50.2% ↓)
		4	59.5	47.0 (21.0% ↓)	83.5	52.9 (36.6% ↓)	130.1	63.9 (50.9% ↓)
		16	90.0	66.3 (26.3% ↓)	130.4	75.9 (41.8% ↓)	209.0	94.8 (54.7% ↓)
	MBPP	1	54.5	45.8 (16.0% ↓)	72.2	49.6 (31.3% ↓)	118.9	60.8 (48.9% ↓)
		4	72.6	57.8 (20.5% ↓)	95.6	63.8 (33.3% ↓)	163.2	79.8 (51.1% ↓)
		16	130.9	101.8 (22.2% ↓)	184.1	116.0 (37.0% ↓)	314.7	143.5 (54.4% ↓)

In Table 23, Fast-dLLM and ACache run on the runtime (Section 4). Even at batch size 1, where Anchor selection is least amortized (Section 5.2), ACache runs at $1.19\times$ and $1.26\times$ Fast-dLLM's throughput on LLaDA and Dream, and at $1.46\times$ and $1.38\times$ BiCache's; at batch sizes 4 and 16 it reaches $1.55\times$ to $1.84\times$ Fast-dLLM's. At batch size 16 it exceeds Section 5.2's $1.71\times$ peak, as this shared prefix is longer than any few-shot affix there (at most 1,016 tokens). BiCache trails even Fast-dLLM: as in Table 6, it runs in its open-source engine, so this compares whole systems.

Table 18: **End-to-end throughput under suffix reuse.** Measured in generated tokens per second.

Model	Dataset	Batch	1-shot		2-shot		4-shot	
			Fast-dLLM	ACache	Fast-dLLM	ACache	Fast-dLLM	ACache
LLaDA	GSM8K	1	104	99 (0.95×)	90	95 (1.06×)	83	88 (1.06×)
		4	263	253 (0.96×)	200	236 (1.18×)	163	212 (1.30×)
		16	393	410 (1.04×)	273	358 (1.31×)	200	307 (1.54×)
	MBPP	1	127	111 (0.87×)	113	138 (1.23×)	87	100 (1.15×)
		4	290	260 (0.90×)	225	292 (1.30×)	161	223 (1.38×)
		16	398	390 (0.98×)	290	390 (1.34×)	191	297 (1.56×)
Dream	GSM8K	1	109	98 (0.90×)	96	98 (1.02×)	90	97 (1.08×)
		4	285	264 (0.93×)	229	250 (1.09×)	183	237 (1.30×)
		16	414	420 (1.01×)	297	375 (1.26×)	222	338 (1.52×)
	MBPP	1	195	178 (0.91×)	175	180 (1.03×)	137	167 (1.22×)
		4	405	383 (0.95×)	319	363 (1.14×)	218	320 (1.47×)
		16	527	533 (1.01×)	386	486 (1.26×)	247	411 (1.66×)

Table 19: **Peak KV cache memory usage (GB) under suffix reuse.**

Model	Dataset	Batch	1-shot		2-shot		4-shot	
			Fast-dLLM	ACache	Fast-dLLM	ACache	Fast-dLLM	ACache
LLaDA	GSM8K	1	0.38	0.62 (66.7% ↑)	0.50	1.12 (125.0% ↑)	0.75	1.38 (83.3% ↑)
		4	1.50	1.38 (8.3% ↓)	2.00	2.00 (0.0%)	3.00	2.50 (16.7% ↓)
		16	6.00	4.38 (27.1% ↓)	8.00	5.12 (35.9% ↓)	12.00	7.00 (41.7% ↓)
	MBPP	1	2.12	2.25 (5.9% ↑)	2.25	2.50 (11.1% ↑)	2.50	3.12 (25.0% ↑)
		4	3.25	3.00 (7.7% ↓)	3.75	3.38 (10.0% ↓)	4.75	4.25 (10.5% ↓)
		16	7.75	6.38 (17.7% ↓)	9.75	7.00 (28.2% ↓)	13.75	8.75 (36.4% ↓)
Dream	GSM8K	1	0.04	0.07 (66.7% ↑)	0.05	0.12 (125.0% ↑)	0.08	0.15 (83.3% ↑)
		4	0.16	0.15 (8.3% ↓)	0.22	0.22 (0.0%)	0.33	0.27 (16.7% ↓)
		16	0.66	0.48 (27.1% ↓)	0.88	0.56 (35.9% ↓)	1.31	0.77 (41.7% ↓)
	MBPP	1	0.23	0.25 (5.9% ↑)	0.25	0.27 (11.1% ↑)	0.27	0.34 (25.0% ↑)
		4	0.36	0.34 (3.8% ↓)	0.41	0.38 (6.7% ↓)	0.52	0.46 (10.5% ↓)
		16	0.85	0.70 (17.7% ↓)	1.07	0.74 (30.8% ↓)	1.50	0.96 (36.4% ↓)

Table 20: **Keeping versus discarding the non-anchor affix cache, 1-shot.** Accuracy (%) on Fast-dLLM. *Anchors only* selects the Anchor Tokens but discards the rest of the affix cache. Bold marks the better value per column. ACache rows repeat Table 5; as there, Avg. omits BABILong suffix.

Model	Ratio	Variant	GSM8K			MBPP			BABILong			Avg.
			Pfx.	Inf.	Suf.	Pfx.	Inf.	Suf.	Pfx.	Inf.	Suf.	
LLaDA	0.2	ACache	**73.6**	**69.9**	**70.3**	**34.9**	**15.4**	**25.1**	99.6	98.9	**76.1**	**61.0**
		Anchors only	71.3	**69.9**	69.1	2.9	6.3	4.9	**99.8**	**99.9**	15.4	53.0
	0.3	ACache	**74.1**	**71.0**	**70.8**	**34.8**	**24.7**	**26.8**	99.7	98.9	**78.6**	**62.6**
		Anchors only	73.1	70.2	**70.8**	5.3	16.2	7.7	**99.8**	**99.7**	56.2	55.3
Dream	0.2	ACache	**70.7**	74.0	70.7	**50.0**	**36.7**	35.9	**95.2**	86.4	**84.5**	64.9
		Anchors only	69.1	**76.7**	**74.8**	45.4	29.6	**49.6**	92.7	**90.6**	64.0	**66.1**
	0.3	ACache	**72.3**	72.1	70.9	**51.3**	**37.2**	**50.7**	**94.9**	87.1	**91.5**	**67.1**
		Anchors only	71.2	**74.2**	**73.5**	49.3	29.1	49.3	91.1	**89.2**	77.8	65.9

Table 21: **Top, random, and bottom Anchor selection, LLaDA, 1-shot.** Accuracy (%) on Fast-dLLM; Top is ACache. Bold marks the best per column and ratio; Avg. omits BABILong suffix.

Ratio	Selection	GSM8K			MBPP			BABILong			Avg.
		Pfx.	Inf.	Suf.	Pfx.	Inf.	Suf.	Pfx.	Inf.	Suf.	
0.2	Top	**73.6**	**69.9**	**70.3**	**34.9**	**15.4**	**25.1**	**99.6**	**98.9**	76.1	**61.0**
	Random	73.2	69.0	69.0	34.1	11.1	18.6	**99.6**	80.4	84.5	56.9
	Bottom	71.6	69.2	68.4	34.4	4.1	8.5	99.1	74.2	**98.1**	53.7
0.3	Top	**74.1**	**71.0**	**70.8**	**34.8**	**24.7**	**26.8**	**99.7**	**98.9**	78.6	**62.6**
	Random	72.0	68.2	69.4	33.8	17.3	23.3	99.6	84.6	79.5	58.5
	Bottom	71.4	68.1	68.0	34.0	4.9	8.5	99.2	74.8	**97.9**	53.6

Table 22: **System-prompt reuse on GSM8K: strict accuracy (%) on Fast-dLLM.** As in BiCache, an answer counts only if the number after "####" matches. BiCache† is the official implementation; Ratio is its mean equivalent Anchor ratio. Bold marks the best reuse method per model; underline, the second-best.

Model	Method	Ratio	Acc.
LLaDA	Direct reuse	0	19.8
	ACache	0.2	59.1
		0.3	<u>65.4</u>
	BiCache†	0.50	**66.9**
	Full recomputation	1	69.0
Dream	Direct reuse	0	78.5
	ACache	0.2	**79.5**
		0.3	<u>78.8</u>
	BiCache†	0.57	70.0
	Full recomputation	1	79.4

Table 23: **System-prompt reuse on GSM8K: end-to-end throughput (tokens/s).** Fast-dLLM is the full recomputation of Table 22; ACache uses Anchor ratio 0.2; BiCache† supports only batch size 1.

Model	Batch 1			Batch 4		Batch 16	
	BiCache†	Fast-dLLM	ACache	Fast-dLLM	ACache	Fast-dLLM	ACache
LLaDA	62	76	91 (1.19×)	138	214 (1.55×)	164	290 (1.77×)
Dream	92	101	126 (1.26×)	167	273 (1.64×)	194	357 (1.84×)