DO WE NEED LEARNED COMPRESSORS? ADAPTING LLMS TO PARAMETER-FREE KV-CACHE COMPRESSION FOR LONG-CONTEXT REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) face growing memory costs during extended reasoning because the key-value (KV) cache expands with every generated token. Recent methods address this by training auxiliary modules to produce compressed representations of past reasoning states. We ask whether the compression function itself needs to be learned. We propose a parameter-free alternative that periodically replaces fixed-stride windows of reasoning tokens with a single KV pair obtained by simply averaging their pre-rotary key and value representations. Rather than optimizing a learned compressor, we freeze the base model and train lightweight LoRA adapters to reason directly from these fixed summary vectors. Across several long-context reasoning benchmarks and compression ratios, our approach performs competitively with learned KV-cache compression baseline while adding zero auxiliary parameters for compression. Our results suggest that effective reasoning over compressed history does not necessarily require the compression operator itself to be learned.

1 INTRODUCTION

Large language models are increasingly capable of solving complex reasoning problems by generating long chains of intermediate steps. Rather than relying only on computation within a single forward pass, they can allocate additional test-time computation by extending their reasoning traces, often improving performance on problems that require multi-step deduction, search, or repeated revision Merrill & Sabharwal (2024); Snell et al. (2024). This ability to reason for longer has become an important source of performance gains, but it also introduces a growing systems cost. During autoregressive generation, each new token produces key and value states at every Transformer layer, which are stored in the KV cache so that subsequent tokens can attend to the full reasoning history. As the generation grows, the cache expands linearly with sequence length, increasing both memory usage and the amount of past context processed at each decoding step. For long reasoning traces, this growing cache becomes a practical bottleneck, limiting how many reasoning steps can be performed under a fixed memory budget and making aggressive test-time scaling increasingly expensive Keles et al. (2023); Monea et al. (2025).

KV-cache compression aims to reduce this cost by retaining only a compact representation of past computation. Training-free methods such as StreamingLLM Xiao et al. (2024) and TOVA Oren et al. (2024) reduce the cache through fixed retention or eviction rules without modifying the model. While effective for general generation, these methods can struggle with multi-step reasoning, where intermediate states that appear unimportant at one point may still become necessary for later deductions. Once such information is discarded, the frozen model has no mechanism to adapt to the altered memory representation or compensate for information that is no longer directly available. Recent training-based approaches therefore adapt the model itself to operate under compression. Breadcrumbs Reasoning (BR), for example, periodically replaces a window of generated states with a learned beacon representation and trains the model to preserve the information needed for subsequent reasoning Monea et al. (2025). This demonstrates that adaptation can make compressed reasoning substantially more effective. However, in such approaches the compressed representation and the model that consumes it are learned together, making it difficult to determine whether suc-

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.

cess comes from learning a better compression function, from adapting the model to compressed memory, or from both.

We ask whether the compression function itself needs to be learned. Our hypothesis is that a simple fixed representation may be sufficient if the model is trained to use it effectively. If so, the main challenge may not be constructing an increasingly specialized summary of past reasoning, but adapting the model to continue reasoning from a reduced representation. Instead of learning a separate compression mechanism, we summarize each reasoning window with a parameter-free operation and place the learning capacity in the model that consumes the compressed state. This allows us to separate the role of the compression rule from the role of adaptation and directly test whether a learned compressor is necessary. In this setup, learning is used to interpret compressed memory rather than to construct it.

In our approach, we independently average the key and value states within each reasoning window and replace the window with a single KV pair. The averaging operator contains no learned parameters, attention-based weighting, or auxiliary compression network. We freeze the base language model and train only lightweight LoRA adapters to operate on the resulting compressed cache, using distillation from an uncompressed reasoning policy. This design separates the construction of compressed memory from the model's adaptation to it: the representation is fixed, while only the model that consumes it is learned.

We evaluate the approach across two complementary forms of long-horizon reasoning: structured algebraic deduction, where intermediate calculations must remain reliable over a relatively fixed sequence of steps, and graph search, where the model may need to explore alternatives and backtrack before reaching a solution. These settings place different demands on compressed memory and allow us to study whether a simple fixed representation remains useful across distinct reasoning patterns. Across a range of compression ratios, we find that the model can continue to reason effectively from averaged KV states when it is explicitly adapted to do so. This suggests that strong reasoning under KV-cache compression does not necessarily require the compression operator itself to be learned.

More broadly, our results point to a different way of thinking about reasoning-time memory compression. Much of the existing effort is devoted to deciding what should be kept, discarded, or encoded into increasingly specialized compressed states. Our results suggest that part of this burden can instead be moved to the model that consumes the compressed memory. A simple representation may be sufficient if the model learns how to interpret it and continue reasoning from it. This suggests that the main learning challenge may lie not in constructing the compressed representation, but in adapting the model to reason effectively from it.

2 Related Work

Training-Free KV Cache Compression. To constrain the linear memory growth of the key-value (KV) cache, a large body of work compresses the cache at inference time without updating model parameters. StreamingLLM Xiao et al. (2024) retains a small set of initial attention-sink tokens together with a sliding window of recent tokens, while H_2O Zhang et al. (2023) dynamically preserves recent tokens and tokens with high cumulative attention. TOVA Oren et al. (2024) instead uses attention scores during autoregressive generation to evict KV entries that receive the least attention. SnapKV Li et al. (2024) uses attention patterns observed near the end of the prompt to select important KV positions for each attention head. Other methods improve how a fixed memory budget is distributed: PyramidKV Cai et al. (2024) varies the retained cache size across Transformer layers, while Ada-KV Feng et al. (2026) allocates budgets adaptively across attention heads. Rather than discarding entries entirely, merging-based approaches such as CaM Zhang et al. (2024) incorporate information from KV entries that would otherwise be evicted into retained representations. While these strategies require zero auxiliary parameters and avoid retraining, they operate entirely on a frozen backbone. In extended multi-step reasoning, where intermediate calculations temporarily receive low attention mass yet remain essential for later deductions, rule-based eviction causes irreversible context loss, while unadapted merging flattens attention distributions and leads to cascading errors.

Learned KV and Context Compression. A complementary line of work adapts the model itself to operate with compressed memory. Gist Tokens Mu et al. (2023) train language models to

route prompt information through a small number of reusable gist representations, while ICAE Ge et al. (2023) learns compact memory slots that summarize longer input contexts. Activation Beacon Zhang et al. (2025) compresses layer-wise key and value activations into learned beacon representations, and Breadcrumbs Reasoning Monea et al. (2025) extends this idea to autoregressive reasoning by periodically replacing windows of generated states with learned beacon representations. Other approaches learn memory-management decisions directly; Neural Garbage Collection Li et al. (2026), for example, trains the model with reinforcement learning to decide which KV entries to evict during reasoning. We shift the learning problem from *how to compress* to *how to use compressed information*. Instead of learning the compression operator, we keep it fixed and adapt the model to interpret the resulting representations effectively.

3 METHOD

We compress each reasoning window into a single key-value pair using a simple parameter-free averaging operation, replacing many cache entries with one compressed entry and thereby reducing memory usage. At the end of each window, we reserve a single cache position for this compressed representation. Lightweight LoRA adapters are then trained to let the model continue reasoning from the resulting cache. The compression rule itself remains fixed throughout training; only the model's adaptation to the compressed cache is learned.

Let W denote the set of entries summarized at a given compression step, and let $K_i^{(\ell)}$ and $V_i^{(\ell)}$ denote the key and value representations of entry i at Transformer layer ℓ, taken after the projection layers and before RoPE is applied. We compute the compressed representation by independently averaging the keys and values within the window:

$$\bar{K}_W^{(\ell)} = \frac{1}{|W|} \sum_{i \in W} K_i^{(\ell)}, \qquad \bar{V}_W^{(\ell)} = \frac{1}{|W|} \sum_{i \in W} V_i^{(\ell)}.$$

The resulting pair $(\bar{K}_W^{(\ell)}, \bar{V}_W^{(\ell)})$ serves as the compressed representation of the window at layer ℓ. It is written into the reserved compression position before the model's standard RoPE operation, so the compressed key inherits the positional encoding of that position. During generation, this single compressed entry replaces the original KV entries of the window, which are then physically evicted from the cache.

Each compression step therefore reduces an entire window of cached representations to one retained entry, with larger windows yielding greater memory savings. The same operation is applied independently at every Transformer layer and introduces no learned parameters, attention-based weighting, or additional compression network; instead, we leave it to the LoRA-adapted model to learn how to interpret the averaged representations and which information to rely on during subsequent reasoning.

Following the windowed structure of Breadcrumbs Reasoning Monea et al. (2025), one cache position is reserved after each compression window. Breadcrumbs fills this position using the KV representation produced by a learned beacon token that attends to the preceding window. In our method, no dedicated beacon token is added to the vocabulary. Instead, we reuse the model's existing end-of-sequence token as a placeholder for the token-shaped forward step. The query and attention output produced at this position are discarded, while the position itself is retained as a slot into which the externally computed averaged key and value are written. The representation that survives in the cache is therefore determined by the fixed averaging operation rather than by the model's self-attention.

The number of entries contributing to each average depends slightly on the position of the window in the generation sequence. The first window contains c entries, where c denotes the nominal compression interval. After compression, the final token generated before the reserved position is re-fed as the first input of the next window so that autoregressive generation can continue correctly. Subsequent windows therefore contain $c+1$ real entries. We account for this by defining the average over the general window W and using $|W|$ as the denominator. In practice, the average is implemented as a masked mean over the valid entries.

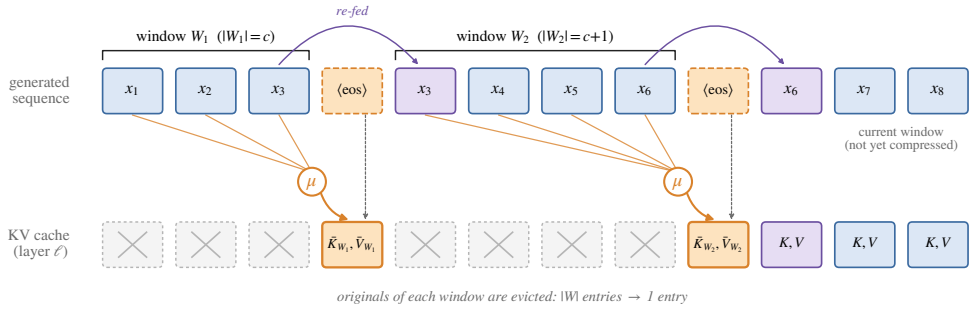


Figure 1: **Window-averaged KV compression.** Generation proceeds in windows of c tokens. At the end of each window W, a reserved $\langle\text{eos}\rangle$ position is appended; its query and attention output are discarded, and only its cache slot is kept. The keys and values of all entries in the window are averaged (μ) independently at every layer ℓ, and the resulting pair $(\bar{K}_W, \bar{V}_W)$ is written into this slot. The original entries of the window then become inaccessible (crossed boxes), so $|W|$ cache entries are replaced by a single one. The last token of each window is re-fed as the first input of the next window (purple), so every window after the first contains $c{+}1$ entries. At inference, the original entries are physically evicted to save memory. During training, the same eviction is simulated with an attention mask, so training stays parallel.

During training, physical cache eviction is simulated through attention masking following the attention-masking scheme of Monea et al. (2025). Tokens after a compression point can access the compressed representation and subsequent entries, but cannot attend to the original entries of the compressed window. This exposes the model to the same information that would remain available after physical eviction, while allowing training to remain parallel and efficient. During autoregressive generation, the original KV entries are instead physically removed and replaced by the single averaged pair. The masked training procedure and physical eviction therefore provide the compressed policy with the same accessible context for computing the distillation objective, while actual eviction realizes the memory savings at inference time.

We next train the model to reason under this compressed context. We first train an uncompressed teacher policy on the reasoning task. This stage allows the teacher to learn how to solve the task using the full KV cache, without any compression-induced information loss. Only after the teacher has been trained to completion do we train the compressed policy. We generate trajectories from the trained teacher and store its predictions, yielding a fixed distillation dataset. Given a teacher trajectory $\mathbf{x} = (x_1, \ldots, x_T)$, we minimize the average token-level KL divergence between the uncompressed teacher policy π_T and the compressed policy π_C:

$$\mathcal{L}(\mathbf{x}) = \frac{1}{T} \sum_{t=1}^{T} D_{\mathrm{KL}}\left(\pi_T(\cdot \mid \mathbf{x}_{<t}) \,\|\, \pi_C(\cdot \mid \mathbf{x}_{<t})\right).$$

For a batch $\mathcal{B}$ of teacher trajectories, the objective becomes

$$\mathcal{L} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathcal{L}(\mathbf{x}).$$

To avoid storing full-vocabulary teacher distributions for every generated token, we retain the teacher's top-100 logits at each step and use the resulting truncated distribution during distillation. The corresponding top-100 tokens consistently account for more than 90% of the teacher probability mass.

The teacher remains fixed throughout distillation. For the compressed policy, the base model is also frozen and only lightweight LoRA parameters are optimized. Consequently, the compression operator itself never learns: every compressed cache entry is produced by the same parameter-free averaging rule, while the LoRA-adapted model learns how to reason from these fixed compressed representations.

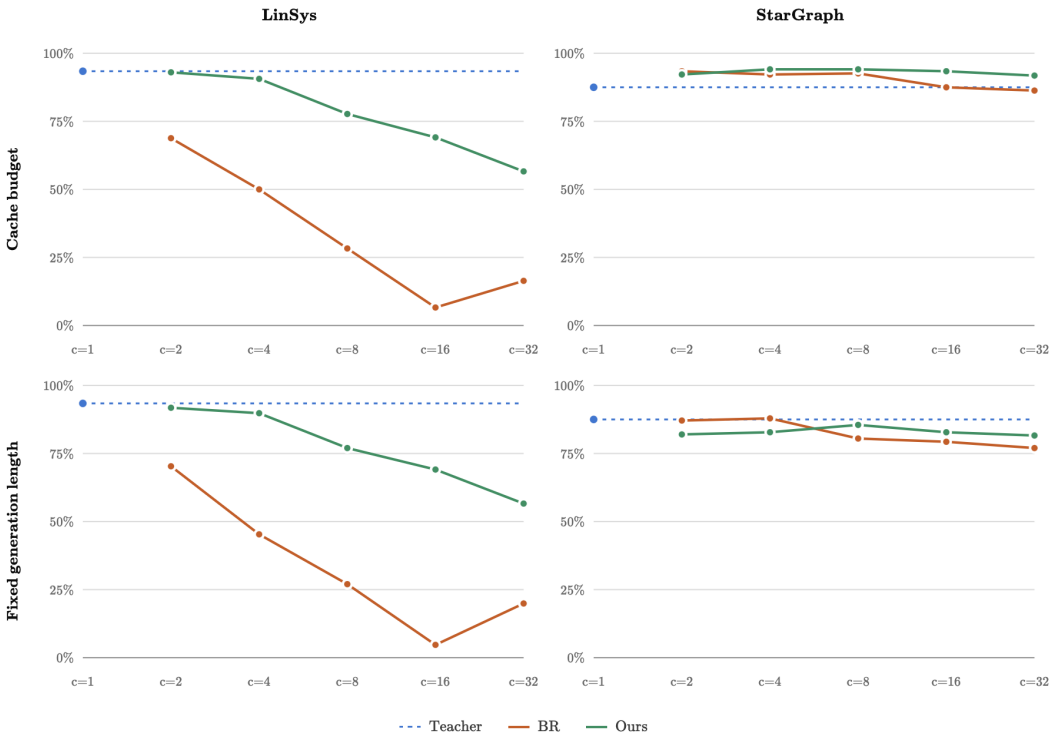


Figure 2: **Accuracy across compression ratios under fixed-cache and fixed-generation evaluation.** Each row reports results on LinSys and StarGraph. The dashed line denotes the uncompressed teacher, while solid lines show the reproduced two-step BR baseline and our method. Under fixed-cache evaluation, compressed models can use saved generation-cache capacity to produce additional reasoning steps within the same memory budget. Under fixed-generation evaluation, all methods receive the same maximum number of generated tokens. Our method degrades substantially more slowly than BR on LinSys, while the two approaches remain closer on StarGraph.

3.1 Implementation Details

We use Qwen2.5-1.5B-Instruct Bai et al. (2023) as the base language model for both the teacher and compressed student policies. The teacher is trained with PPO Schulman et al. (2017) using the `verl` Sheng et al. (2024) framework with FSDP and vLLM-backed rollouts: actor learning rate 1×10^{-6}, critic learning rate 1×10^{-5}, PPO mini-batch size 256, 1 PPO epoch per batch, clip ratio 0.2, entropy coefficient 1×10^{-3}, and GAE with $\gamma = 1.0, \lambda = 1.0$. We train all models for 1000 steps. We use a 1000-token response budget for LinSys and 2000 tokens for StarGraph, as the teacher did not reliably learn the task under the 1000-token limit. Teacher training is distributed across four NVIDIA L40S GPUs, with the actor, critic, and rollout generation. The compressed student is trained on a single L40S GPU. The base model is kept frozen, and only a LoRA Hu et al. (2021) adapter (rank 16, $\alpha = 32$, applied to all attention and MLP projections) is optimized via Hugging Face `peft` Mangrulkar et al. (2022), with learning rate 1×10^{-3} and batch size 256.

4 Experiments

4.1 Experimental Setup

Tasks. We evaluate our method on LinSys and StarGraph Bachmann & Nagarajan (2024), for which the datasets used by Breadcrumbs Reasoning are publicly available.[1] We exclude Count-

[1] LinSys: `https://huggingface.co/datasets/giovannimonea/linear_systems_vars4_abs20_nnz2`; StarGraph: `https://huggingface.co/datasets/giovannimonea/star_graphs_paths25_len5_nodes1000`.

	LinSys						StarGraph					
Compr. Ratio	1	2	4	8	16	32	1	2	4	8	16	32
No Compression												
Teacher	0.934	–	–	–	–	–	0.875	–	–	–	–	–
Training-Based Methods												
BR	–	0.688	0.500	0.283	0.066	0.164	–	**0.934**	0.922	0.926	0.875	0.863
Ours	–	**0.930**	**0.906**	**0.777**	**0.691**	**0.566**	–	0.922	**0.941**	**0.941**	**0.934**	**0.918**
Training-Free Methods												
SnapKV	–	0.004	0.000	0.000	0.000	0.000	–	0.199	0.121	0.086	0.059	0.078
TOVA	–	0.004	0.004	0.000	0.000	0.000	–	0.180	0.148	0.094	0.066	0.086
StreamingLLM	–	0.719	0.164	0.133	0.094	0.051	–	0.441	0.387	0.254	0.164	0.215

Table 1: Accuracy under a fixed generation-cache budget on LinSys and StarGraph. The $1\times$ column reports the shared uncompressed teacher used by all methods. We use a maximum cache budget of $1{,}000$ entries for LinSys and $2{,}000$ entries for StarGraph, matching the corresponding teacher training budgets. BR and our method use the same teacher trajectories under two-step distillation.

down from the main comparison because the exact evaluation instances are not released, and our reconstruction produced substantially different uncompressed teacher performance. We consider compression ratios $c \in \{2, 4, 8, 16, 32\}$, where $c = 1$ denotes the uncompressed teacher. LinSys evaluates structured multi-step deduction over systems of linear equations, while StarGraph requires recovering a path from a long graph description.

Training-based baseline. Our primary training-based baseline is Single-Ratio Breadcrumbs Reasoning (SR BR) Monea et al. (2025). We reproduce SR BR using the training configuration reported in the original work. For the controlled comparison, we use the two-step distillation setting: the uncompressed teacher is first trained to completion, after which both BR and our method are distilled from the same fixed teacher checkpoint and teacher trajectories. This controls the teacher, distillation data, and training budget across the two methods. All experiments use Qwen2.5-1.5B-Instruct Bai et al. (2023) as the base language model for both the uncompressed teacher and compressed policies.

Training-free baselines. We additionally compare against training-free KV-cache compression methods that operate on the same uncompressed teacher without parameter updates: StreamingLLM Xiao et al. (2024), TOVA Oren et al. (2024), and SnapKV Li et al. (2024). Following the memory-matching protocol of BR, the generation cache is allowed to grow by one retained entry every c generated tokens. StreamingLLM retains the complete input question. For TOVA and SnapKV, the initial cache budget is set to the question length plus c, so that compression begins only after at least c generated tokens.

Evaluation protocols. We evaluate all methods on 256 examples from the `eval` split using seed 0, under two inference settings. In the *fixed-cache* setting, generation is constrained to a maximum generation KV-cache budget of $1{,}000$ entries for LinSys and $2{,}000$ entries for StarGraph. The KV states corresponding to the input prompt are not included in this budget. Compressed models can therefore use the saved generation-cache capacity to produce longer reasoning traces. In the *fixed-generation* setting, all methods are instead evaluated with the same maximum number of generated tokens. This removes the additional generation length enabled by compression.

4.2 Results

Fixed-cache performance. Table 1 compares the methods under a fixed KV-cache budget. On LinSys, our method consistently outperforms BR across all compression ratios. At $2\times$ compression, our method nearly matches the uncompressed teacher (0.930 vs. 0.934), while BR drops to 0.688. The gap widens as compression becomes more aggressive: at $8\times$, our method achieves 0.777 compared with 0.283 for BR, and at $32\times$ it retains 0.566 accuracy compared with 0.164 for BR. These results show that our fixed-averaging approach preserves substantially more reasoning performance than the reproduced BR baseline when both methods are distilled from the same teacher and trajectories.

	LinSys						StarGraph					
Compr. Ratio	1	2	4	8	16	32	1	2	4	8	16	32
No Compression												
Teacher	0.934	–	–	–	–	–	0.875	–	–	–	–	–
Training-Based Methods												
BR	–	0.703	0.453	0.270	0.047	0.199	–	**0.871**	**0.879**	0.805	0.793	0.770
Ours	–	**0.918**	**0.898**	**0.770**	**0.691**	**0.566**	–	0.820	0.828	**0.855**	**0.828**	**0.816**
Training-Free Methods												
SnapKV	–	0.004	0.000	0.000	0.000	0.000	–	0.195	0.109	0.082	0.055	0.078
TOVA	–	0.004	0.004	0.000	0.000	0.000	–	0.168	0.133	0.090	0.062	0.074
StreamingLLM	–	0.641	0.070	0.008	0.000	0.000	–	0.402	0.328	0.219	0.152	0.195

Table 2: Accuracy at a fixed generation length on LinSys and StarGraph. The $1\times$ column reports the shared uncompressed teacher used by all methods. We use a maximum generation length of $1{,}000$ tokens for LinSys and $2{,}000$ for StarGraph.

The difference is smaller on StarGraph, where both methods retain high accuracy. BR performs slightly better at $2\times$, while our method matches or exceeds BR at several higher compression ratios. Notably, our method reaches 0.941 accuracy at $4\times$ compression, exceeding the uncompressed teacher's 0.875. Under a fixed cache budget, however, compressed models can use the freed cache capacity to generate longer reasoning traces than the uncompressed teacher. This is particularly beneficial on StarGraph, where solving harder instances may require additional search and backtracking. The result therefore reflects both the quality of the compressed representation and the additional test-time reasoning enabled by compression.

Fixed-generation performance. The fixed-generation setting provides a complementary view of compression by removing the additional reasoning length that may be enabled by a smaller KV cache. This distinction matters differently across the two tasks. LinSys follows a relatively fixed deterministic solution procedure, and its response length remains nearly constant across compression ratios. Consistent with this behavior, performance under fixed-generation and fixed-cache evaluation differs only marginally, suggesting that additional generation provides little benefit on this task.

StarGraph behaves differently. Solving the task requires searching through candidate paths and potentially backtracking after unsuccessful branches, so the amount of reasoning required can vary substantially across instances. Under a fixed cache budget, compression can therefore provide an additional benefit by allowing the model to continue exploring for more steps. Fixed-generation evaluation removes this advantage and instead measures how effectively the model can reason from compressed history within a common search budget. Under this setting, both BR and our method remain relatively close to the uncompressed teacher across compression ratios, with only modest differences between the two approaches. BR is slightly stronger at some lower compression ratios, while our method is stronger at several more aggressive settings. Overall, the similarity in performance suggests that both learned beacon compression and fixed mean pooling can preserve enough information to support the StarGraph search process, even though they construct the compressed representation in fundamentally different ways.

Task-dependent behavior. The pronounced difference on LinSys is consistent with the failure analysis reported by Breadcrumbs Reasoning Monea et al. (2025). Monea et al. find that BR's performance degradation on LinSys is driven primarily by arithmetic errors rather than failures to memorize or retrieve compressed information, and that these errors increase with stronger compression. They suggest that numerical information may be particularly difficult to compress and that the compression procedure may interfere with arithmetic computation.

Our method exhibits substantially less degradation on the same task despite using a fixed, parameter-free compressor. This suggests that learning the compressed representation itself is not necessary for preserving structured numerical reasoning. One possible explanation is that freezing the base model and adapting only lightweight LoRA parameters better preserves the teacher's existing reasoning behavior while learning how to consume the compressed states.

Comparison with reported BR results. Our main comparison uses the reproduced two-step BR baseline because it shares the same teacher and distillation data as our method. For context, the original BR paper reports stronger SR BR results under joint RL-distillation. On LinSys, our method also exceeds these reported results across all compression ratios. For example, under the fixed cache budget setting, our method achieves 0.777 and 0.566 accuracy at $8\times$ and $32\times$ compression, compared with 0.410 and 0.297 reported for joint SR BR. The same pattern holds under fixed generation, where our method obtains 0.770 and 0.566, compared with 0.395 and 0.289 for joint SR BR. The published BR experiments use a separately trained teacher with 0.918 LinSys accuracy, compared with 0.934 in our setup, so we treat these numbers as supporting reference rather than a controlled head-to-head comparison.

Training-free baselines. Training-free compression performs substantially worse than both training-based approaches. On LinSys, TOVA collapses to near-zero accuracy across compression ratios, while StreamingLLM degrades sharply as the cache is reduced. The same trend appears on StarGraph, where all baselines remain well below the adapted compressed policies. These results indicate that compression alone is insufficient for long reasoning: the model must learn to operate under the altered memory representation. Importantly, this adaptation does not require the compression operator itself to be learned.

KV-cache memory scaling. For Qwen2.5-1.5B-Instruct in bf16, each cached token occupies 28,672 bytes. The effective memory reduction depends on how much of the cache comes from generated reasoning versus the uncompressed prompt. On representative runs, a LinSys example at $c = 4$ reduces the generated cache by $3.99\times$ and the total cache by $3.56\times$, while a StarGraph example at $c = 32$ reduces the generated cache by $26.83\times$ and the total cache by $1.58\times$. The larger gap on StarGraph reflects its much longer prompt, showing that end-to-end memory savings depend on the prompt-to-generation ratio in addition to the nominal compression rate.

4.3 Ablation Studies

Ablating adaptation and the compressed representation. We examine separately whether adaptation is necessary and whether performance depends on how the retained compressed state is constructed. All experiments use LinSys at $c = 4$ under fixed cache budget evaluation. To isolate the role of adaptation, we first keep mean pooling but remove LoRA. The frozen model fails to make effective use of the averaged KV states: accuracy drops from 90.6% to 0%, generations become very short, approximately 121–128 tokens on average, and typically collapse into malformed or incoherent outputs rather than complete derivations. This shows that the mean-pooled representation is not directly usable by the unadapted base model and that explicit adaptation to the compressed cache is necessary.

We next hold the LoRA setup fixed and vary the retained representation. In the self-attended-slot variant, a separately trained LoRA policy uses the same eviction schedule but no external mean-pool overwrite; the reserved position instead retains the KV state produced by its own self-attention. This variant also reaches 0% accuracy. Generations remain normal in length, averaging 759.6 tokens, and usually contain plausible step-by-step algebra, but fail through incorrect intermediate calculations and occasional repetition.

As a second representation control, we train another LoRA policy that, instead of averaging the KV states within each window, retains only the KV state of the final token. This variant uses the same one-entry memory budget, compression schedule, and LoRA configuration as our method, but replaces averaging with a simple token-selection rule. It also achieves 0% accuracy. Its failure mode is qualitatively different from the self-attended-slot variant: all 256 evaluation examples reach the 1,000-token generation limit without producing a correct answer, with generations frequently entering repetitive substitution loops rather than converging to a solution. The same instability is visible during training. Whereas the mean-pool model shows a smooth reduction in distillation loss, the last-token variant remains noisy through the end of training, with substantial step-to-step loss variation and large gradient-norm spikes.

Together, these results show that adaptation is necessary for making use of the averaged cache representation, but that LoRA adaptation alone does not make arbitrary one-entry summaries effective. Under the same adaptation configuration and compression budget, mean pooling reaches 90.6%

LoRA Targets	Accuracy (%)
Uncompressed teacher	93.4
All attention + MLP projections	90.6
Gate only	85.6
Attention only	82.0
MLP only	78.1

Table 3: Ablation of LoRA target modules under mean-pool compression with $c = 4$. The uncompressed teacher is included for reference.

accuracy, whereas both the self-attended-slot and last-token variants fail to learn successful compressed policies.

Which parameters need to adapt? We next study which parts of the model are responsible for adapting to the compressed KV representation. All experiments use mean-pool compression with $c = 4$. The full configuration applies LoRA to the attention projections (`q_proj`, `k_proj`, `v_proj`, and `o_proj`) and the MLP projections (`gate_proj`, `up_proj`, and `down_proj`).

Restricting LoRA to any subset of modules reduces performance relative to the full configuration. Among the restricted variants, adapting only `gate_proj` performs best at 85.6%, followed by the attention-only configuration at 82.0% and the MLP-only configuration at 78.1%. The strongest performance is obtained when both attention and MLP components are adapted, suggesting that effective reasoning from compressed KV states benefits from changes across multiple parts of the model rather than a single projection family.

5 Limitations

Our results also expose limitations of reasoning under aggressive KV-cache compression. On LinSys, performance declines as the compression ratio increases, and the remaining failures are often arithmetic slips within otherwise coherent derivations. Mean-pool adaptation therefore reduces the degradation observed under compression but does not eliminate the sensitivity of numerical reasoning to information loss. KV-cache compression is also most beneficial when memory limits how long the model can continue reasoning. When generation length is fixed independently of memory, compression still reduces memory usage but cannot provide additional reasoning steps. This is particularly relevant for search-based tasks such as StarGraph, where some difficult instances may require more exploration than the fixed response budget allows. Finally, it remains unclear how well the same behavior extends across other reasoning task types, such as logical reasoning, planning, code reasoning, and multi-hop question answering.

6 Conclusion

We study whether effective KV-cache compression for long reasoning requires learning the compression function itself. Instead of learning a specialized compressed representation, we average the key and value states within each reasoning window and adapt lightweight LoRA parameters to reason from the resulting compressed memory. Across structured algebraic reasoning and graph-search tasks, this simple approach remains effective across compression ratios and is competitive with, and in some settings stronger than, learned KV compression. Our ablations show that averaging alone is not sufficient: the model must learn how to use the compressed representation. Replacing the averaged state with a self-computed one-slot representation under the same compression schedule also fails, showing that both the retained representation and the adaptation to it matter. These results suggest that reasoning-time KV compression does not necessarily require a learned compression operator. A simple representation can be sufficient when the model is explicitly adapted to use it, motivating further study of how models can reason reliably from compact memory across broader tasks.

Reproducibility Statement. We provide the information needed to reproduce our experiments throughout the paper. Section 3 describes the KV-cache compression procedure, training-time masking, and distillation objective, together with the model, LoRA, and training configuration. Section 4.1 specifies the datasets, including links to the public LinSys and StarGraph data, the evaluation split and seed, baseline implementations, compression ratios, and evaluation protocols. The appendix provides additional implementation details and representative prompts and model generations to clarify the experimental setting.

AI Use Statement. Generative AI tools were used only to assist with writing, grammar, and language refinement.

REFERENCES

Gregor Bachmann and Vaishnavh Nagarajan. The pitfalls of next-token prediction. *arXiv preprint arXiv:2403.06963*, 2024.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, et al. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*, 2024.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S Kevin Zhou. Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference. *Advances in Neural Information Processing Systems*, 38:113152–113188, 2026.

Tao Ge, Jing Hu, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model. *arXiv preprint arXiv:2307.06945*, 2023.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Feyza Duman Keles, Pruthuvi Mahesakya Wijewardena, and Chinmay Hegde. On the computational complexity of self-attention. In *International conference on algorithmic learning theory*, pp. 597–619. PMLR, 2023.

Michael Y Li, Jubayer Ibn Hamid, Emily B Fox, and Noah D Goodman. Neural garbage collection: Learning to forget while learning to reason. *arXiv preprint arXiv:2604.18002*, 2026.

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. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024.

Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, Benjamin Bossan, and Marian Tietz. PEFT: State-of-the-art parameter-efficient fine-tuning methods. `https://github.com/huggingface/peft`, 2022.

William Merrill and Ashish Sabharwal. The expressive power of transformers with chain of thought. In *International Conference on Learning Representations*, volume 2024, pp. 7690–7706, 2024.

Giovanni Monea, Yair Feldman, Shankar Padmanabhan, Kianté Brantley, and Yoav Artzi. Breadcrumbs reasoning: Memory-efficient reasoning with compression beacons. *arXiv preprint arXiv:2510.13797*, 2025.

Jesse Mu, Xiang Li, and Noah Goodman. Learning to compress prompts with gist tokens. *Advances in Neural Information Processing Systems*, 36:19327–19352, 2023.

Matanel Oren, Michael Hassid, Nir Yarden, Yossi Adi, and Roy Schwartz. Transformers are multi-state rnns. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 18724–18741, 2024.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, volume 2024, pp. 21875–21895, 2024.

Peitian Zhang, Zheng Liu, Shitao Xiao, Ninglu Shao, Qiwei Ye, and Zhicheng Dou. Long context compression with activation beacon. In *International Conference on Learning Representations*, volume 2025, pp. 101932–101948, 2025.

Yuxin Zhang, Yuxuan Du, Gen Luo, Yunshan Zhong, Zhenyu Zhang, Shiwei Liu, and Rongrong Ji. Cam: Cache merging for memory-efficient llms inference. In *Forty-first international conference on machine learning*, 2024.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in neural information processing systems*, 36:34661–34710, 2023.

A Appendix

A.1 Qualitative Examples

We provide representative successful and unsuccessful generations from our compressed models on both LinSys and StarGraph. For each task, we include one correct and one incorrect example to illustrate the qualitative behavior of the model under compression. The LinSys examples highlight how arithmetic or algebraic errors can propagate through otherwise structured derivations, while the StarGraph examples illustrate successful path recovery as well as failures caused by incorrect graph transitions. Model outputs preserve the original generated content, with formatting normalized for readability.

A.1.1 LinSys

Correct example

Setting. Ours, $c = 32$, fixed-generation evaluation.

Prompt.

```
Solve the following system of linear equations:
-13*x2 + 11*x3 = -59
-11*x1 + 4*x4 = 148
-4*x3 + 18*x4 = 32
x3 + 19*x4 = 86

Find the values for x1, x2, x3, x4. Make sure to solve it by
thinking step by step, and do not assume access to any external tools.
Return the final answer as a list of numbers in <answer></answer>
tags, for example <answer>[1, -2, 3, 7]</answer>.
```

Model output.

```
To solve the system of linear equations, we will use the method of
substitution and elimination. The given system of equations is:

1. -13x_2 + 11x_3 = -59
2. -11x_1 + 4x_4 = 148
3. -4x_3 + 18x_4 = 32
4. x_3 + 19x_4 = 86

First, let's solve equation (4) for x_3:
x_3 + 19x_4 = 86
x_3 = 86 - 19x_4

Now substitute x_3 into equation (3):
-4(86 - 19x_4) + 18x_4 = 32
-344 + 76x_4 + 18x_4 = 32
94x_4 - 344 = 32
94x_4 = 376
x_4 = 4

Now substitute x_4 = 4 into equation (4):
x_3 + 19(4) = 86
x_3 + 76 = 86
x_3 = 10

Now substitute x_3 = 10 into equation (1):
-13x_2 + 11(10) = -59
-13x_2 + 110 = -59
-13x_2 = -169
x_2 = 13

Now substitute x_2 = 13 into equation (2):
-11x_1 + 4(4) = 148
-11x_1 + 16 = 148
```

```
-11x_1 = 132
x_1 = -12

Now substitute x_1 = -12 into equation (3):
-4(10) + 18(4) = 32
-40 + 72 = 32

This is true, so our values for x_1, x_2, x_3, x_4 are correct.

So the solution is:
<answer>[ -12, 13, 10, 4 ]</answer>
```

Reference answer.

$$[-12, 13, 10, 4]$$

Outcome. Correct.

Incorrect example

Setting. Ours, $c = 32$, fixed-generation evaluation.

Prompt.

```
Solve the following system of linear equations:
12*x2 - 19*x4 = -253
3*x1 - 15*x4 = -114
8*x3 + 15*x4 = 193
-9*x3 - 17*x4 = -218
Find the values for x1, x2, x3, x4. Make sure to solve it by thinking
step by step, and do not assume access to any external tools.
Return the final answer as a list of numbers in <answer> </answer>
tags, for example <answer>[1, -2, 3, 7]</answer>.
```

Model output.

```
To solve the system of linear equations, we will use the method of
substitution and elimination. The given system of equations is:

1. 12x_2 - 19x_4 = -253
2. 3x_1 - 15x_4 = -114
3. 8x_3 + 15x_4 = 193
4. -9x_3 - 17x_4 = -218

First, let's solve equation (4) for x_3:
-9x_3 - 17x_4 = -218
-9x_3 = 17x_4 - 218
x_3 = -(17x_4 - 218) / 9

Now substitute x_3 into equation (3):
8(-(17x_4 - 218) / 9) + 15x_4 = 193
-(136x_4 - 1744) / 9 + 15x_4 = 193
-136x_4 + 1744 + 135x_4 = 1747
-11x_4 + 1744 = 1747
-11x_4 = -11
x_4 = 1

Now substitute x_4 = 1 into equation (1):
12x_2 - 19(1) = -253
12x_2 - 19 = -253
12x_2 = -234
x_2 = -19

Now substitute x_2 = -19 into equation (2):
3x_1 - 15(1) = -114
3x_1 - 15 = -114
3x_1 = -99
```

```
x_1 = -33

Now substitute x_1 = -33 into equation (4):
-9x_3 - 17(1) = -218
-9x_3 - 17 = -218
-9x_3 = -201
x_3 = 22

Now substitute x_3 = 22 into equation (3):
8(22) + 15(1) = 193
176 + 15 = 193

This is true, so our values for x_1, x_2, x_3, x_4 are correct.

So the solution is:
<answer>[ -33, -19, 22, 1 ]</answer>
```

Reference answer.

$$[-3, -10, 11, 7]$$

Outcome. Incorrect. The model completes the derivation normally but makes arithmetic errors in the intermediate calculations, which propagate to an incorrect final answer.

A.1.2 STARGRAPH

Correct example

Setting. Ours, $c = 32$, fixed-generation evaluation.

Prompt.

```
You are given a star graph with the following nodes: 3, 61, 88, 178,
↪  281, 564, 736, 784, 801, 958, 979.
The graph has the following directed edges:
88 -> 281
979 -> 784
958 -> 3
801 -> 564
281 -> 736
736 -> 178
564 -> 979
784 -> 958
801 -> 61
61 -> 88
Find the path from the center node 801 to the target node 3.
Think step by step about the graph structure and trace the path from
↪  the center node to the target node.
Return your answer as a list of nodes representing the path from
↪  center to target. Return the final answer in <answer> </answer>
↪  tags, for example <answer>[1, 3, 7, 12]</answer>.
```

Model output.

```
To find the path from the center node 801 to the target node 3, we
need to
follow the directed edges:

1. Start at node 801
2. Follow 801 -> 564
3. Follow 564 -> 979
4. Follow 979 -> 784
5. Follow 784 -> 958
6. Follow 958 -> 3

So, the path from 801 to 3 is:
```

```
801 -> 564 -> 979 -> 784 -> 958 -> 3

The final answer is:
<answer>[801, 564, 979, 784, 958, 3]</answer>
```

Reference answer.

$$[801, 564, 979, 784, 958, 3]$$

Outcome. Correct.

Incorrect example

Setting. Ours, $c = 32$, fixed-generation evaluation.

Prompt.

```
You are given a star graph with the following nodes: 2, 39, 64, 124,
↪  149, 196, 244, 280, 290, 294, 338, 362, 405, 419, 467, 470, 472,
↪  495, 545, 569, 624, 673, 674, 678, 749, 771, 779, 831, 853, 861,
↪  921.
The graph has the following directed edges:
470 -> 280
244 -> 294
405 -> 470
124 -> 290
624 -> 779
467 -> 64
196 -> 419
2 -> 624
749 -> 569
472 -> 673
771 -> 405
196 -> 244
362 -> 831
196 -> 472
569 -> 545
853 -> 124
196 -> 853
196 -> 861
419 -> 2
921 -> 362
779 -> 495
290 -> 678
196 -> 467
678 -> 674
64 -> 39
673 -> 749
149 -> 338
39 -> 149
861 -> 771
294 -> 921
Find the path from the center node 196 to the target node 495.
Think step by step about the graph structure and trace the path from
↪  the center node to the target node.
Return your answer as a list of nodes representing the path from
↪  center to target. Return the final answer in <answer> </answer>
↪  tags, for example <answer>[1, 3, 7, 12]</answer>.
```

Model output.

```
To find the path from the center node 196 to the target node 495, we
need to
follow the directed edges:

1. Start at node 196
2. Follow 196 -> 472
```

```
3. Follow 472 -> 673
4. Follow 673 -> 749
5. Follow 749 -> 569
6. Follow 569 -> 545
7. Follow 545 -> 196 (but we already followed this path)

We need to find a new path from 196 to 495. We can follow:

8. Follow 196 -> 853
9. Follow 853 -> 124
10. Follow 124 -> 290
11. Follow 290 -> 678
12. Follow 678 -> 674
13. Follow 674 -> 196 (but we already followed this path)

We need to find a new path from 196 to 495. We can follow:

14. Follow 196 -> 467
15. Follow 467 -> 64
16. Follow 64 -> 39
17. Follow 39 -> 149
18. Follow 149 -> 338
19. Follow 338 -> 196 (but we already followed this path)

We need to find a new path from 196 to 495. We can follow:

10. Follow 196 -> 861
11. Follow 861 -> 771
12. Follow 771 -> 405
13. Follow 405 -> 470
14. Follow 470 -> 280
15. Follow 280 -> 196 (but we already followed this path)

We need to find a new path from 196 to 495. We can follow:

11. Follow 196 -> 244
12. Follow 244 -> 294
13. Follow 294 -> 921
14. Follow 921 -> 362
15. Follow 362 -> 831
16. Follow 831 -> 196 (but we already followed this path)

We need to find a new path from 196 to 495. We can follow:

12. Follow 196 -> 472 (but we already followed this path)

We need to find a new path from 196 to 495. We can follow:

13. Follow 196 -> 779
14. Follow 779 -> 495

So, the final path from 196 to 495 is:
196 -> 779 -> 495

The answer is:
<answer>[196, 779, 495]</answer>
```

Reference answer.

$$[196, 419, 2, 624, 779, 495]$$

Outcome. Incorrect. The model follows several valid partial branches but repeatedly hallucinates edges returning to the source node. It eventually also hallucinates the edge $196 \to 779$; although $779 \to 495$ is present in the graph, the resulting path is invalid.