Cacheback: Speculative Decoding With Nothing But Cache

Zhiyao Ma* **and In Gim*** **and Lin Zhong**
Yale University
{zhiyao.ma,in.gim,lin.zhong}@yale.edu

Abstract

We present *Cacheback Decoding*, a training-free and model-agnostic speculative decoding method that exploits the locality in language to accelerate Large Language Model (LLM) inference. Cacheback leverages only Least Recently Used (LRU) cache tables of token n-grams to generate draft sequences. Cacheback achieves state-of-the-art performance among comparable methods despite its minimalist design, and its simplicity allows easy integration into existing systems. Cacheback also shows potential for fast adaptation to new domains.

1 Introduction

Cache Language Models (CLMs), notable innovations from the 1990s (Kuhn and De Mori, 1990), enhanced the predictive capabilities of n-gram models. They store recently observed n-grams in a cache. Using the cache table, they modify the probabilities assigned by the base n-gram model to favor n-grams present in the cache, effectively exploiting the linguistic phenomenon of "burstiness," i.e., the increased likelihood of recently used words reappearing.

With the subsequent rise of Large Language Models (LLMs), whose massive parameterization enables them to capture complex, long-distance contextual patterns, the original purpose of CLMs appears to have been superseded. However, we re-examine the utility of caching not to improve the intrinsic modeling power of already potent LLMs, but as a surprisingly effective tool for a different objective: accelerating LLM generative inference.

We present *Cacheback Decoding*, a novel method that repurposes the CLM concept for the modern challenge of Speculative Decoding (SD). In the SD framework (Leviathan et al., 2023), a faster mechanism proposes a sequence of draft tokens, which the LLM then attempts to validate in a single forward pass, potentially accepting multiple tokens at once and thereby reducing overall latency. Cacheback generates these drafts without auxiliary neural models or complex algorithmic procedures.

Cacheback's drafting mechanism is extremely simple: It maintains a cache table with the Least Recently Used (LRU) eviction policy. This table maps a tuple of leading tokens to a set of tuples of immediately following tokens most recently observed after the leading ones in the ongoing generation process or recent context. Cacheback generates a tree of draft tokens by recursively querying the cache table using the last few tokens in a tree branch as the key and retrieving the follower tokens to grow the tree. This draft generation step is lightweight, typically executing in microseconds, thus imposing negligible overhead on the decoding loop.

Our empirical evaluations on the SpecBench benchmark (Xia et al., 2024) demonstrate that Cacheback, despite its minimalist design, achieves state-of-the-art performance in wall-clock speedup and token acceptance ratio among comparable baselines that do not require draft model training or model architecture modifications. The effectiveness of Cacheback suggests avenues for future work, including dynamic cache scaling and rapid domain adaptation for draft generation, a traditional strength of CLMs.

2 Background and Related Work

2.1 Exploiting Locality in Language Modeling

The principle of locality, referring to the tendency for related words to appear in close proximity, is a universal characteristic of both artificial (e.g., programming languages) and natural languages. This phenomenon is theoretically grounded in information theory. As Futrell et al. (2015) posits, if we consider the limitations of human information processing and the constraints of short-term memory, then efficient languages are expected to favor lo-

*Equal contribution.

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 31068–31073

cal information structures. That is, *words that are linked in meaning or usage should occur near each other*. This inherent locality can be effectively exploited using surprisingly simple mechanisms like caching. CLMs were pioneering in this regard, integrating n-gram caches to enhance language modeling quality by re-weighting probabilities from a base n-gram model. Despite their simplicity, CLMs demonstrated significant empirical efficacy, achieving perplexity reductions of 38% to 50% in tasks like speech recognition (Jelinek et al., 1991; Clarkson and Robinson, 1997), which led to their popularity in the 1990s.

2.2 Speculative Decoding

Speculative Decoding (SD) is a lossless method to accelerate LLM inference by using a faster draft mechanism, or drafter, to predict future tokens (Leviathan et al., 2023). The LLM then validates these candidates in one forward pass in parallel, potentially accepting multiple tokens at once to reduce generation latency. SD methods vary based on whether the approach requires a specific model (model-dependent vs. model-agnostic) and whether the drafter requires training (training-required vs. training-free). Some SD approaches use an off-the-shelf smaller model in a model family as the drafter (Xia et al., 2023) and thus are model-dependent and training-free. DistillSpec (Zhou et al., 2024) trains a distilled drafter given any model, so the approach is model-agnostic but training-required. Methods like EAGLE (Li et al., 2024) and MEDUSA (Cai et al., 2024) modify the LLM by adding auxiliary components, making them model-dependent and training-required. Recently, SD methods that are both model-agnostic and training-free are favored for their plug-and-play convenience and broad applicability. They often leverage heuristics, small pre-trained models, or prompt and history information, such as lookahead strategies (Zhao et al., 2024; Fu et al., 2024), prompt-based lookups (Saxena, 2023), or online draft construction (Liu et al., 2024).

3 Cacheback Decoding

We propose Cacheback, a simple yet effective speculative decoding method that leverages cache tables to exploit the locality in language for speedup. The table caches previously seen n-grams, divided into a *leader* part and a *follower* part (§3.1). When queried with a leader, the table returns a list of followers that have previously appeared immediately after the leader. At each decoding step, Cacheback generates a tree of draft tokens by recursively querying the cache table and verifies them in parallel with one forward pass of the LLM (§3.2). Our strategy follows the intuition that n-grams which have recently appeared are likely to reappear. After the LLM forward pass, Cacheback updates the cache table to include new n-grams from accepted tokens, evicting stale entries with the least recently used (LRU) policy if necessary. To avoid the cold-start problem, Cacheback initializes the cache table with frequent n-grams observed in large training corpora (§3.3). Despite being a simple method, Cacheback achieves superior or comparable performance to many sophisticated model-agnostic and training-free methods developed in recent years (§4).

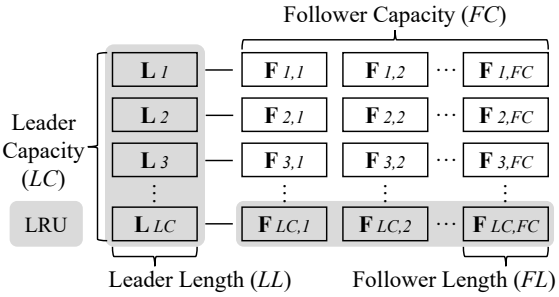

Figure 1: Cacheback's cache table structure. Each leader is associated with a list of followers. Entries are evicted using the least recently used (LRU) policy.

We have open-sourced the implementation of Cacheback with integration into the SpecBench benchmark suite.[1] We have also hosted the binary artifacts, i.e., frozen cache tables (§3.3), on Hugging Face for public access.[2]

3.1 Cache Table Structure

As shown in Figure 1, the cache table has a simple structure in which the leaders and followers are both tuples of tokens (i.e., n-grams) and can be of different lengths, denoted by the leader length (LL) and follower length (FL). The table associates each leader with a list of followers. When queried with a leader, the table returns the associated followers, or an empty list if the leader is absent. The maximum number of leaders in a table and the

[1] https://github.com/zyma98/Spec-Bench/tree/cacheback

[2] https://huggingface.co/datasets/zyma98/cacheback_openwebtext_sample_100

maximum number of followers per leader are denoted by the leader capacity (LC) and the follower capacity (FC), respectively.

The table updates its entries by accepting a leader-follower pair. The table first checks if the leader exists in the table. If not, the table creates a new entry for the leader and initializes the follower list with the new follower. Otherwise, the table appends the new follower to the existing follower list if the follower is not already present.

When the number of leaders exceeds LC, the table evicts the least recently used (LRU) leader and its followers upon inserting a new leader. Both querying and inserting a leader update that leader's recency. Likewise, if the number of followers of a leader exceeds FC, the table evicts the leader's LRU follower. A notable difference is that the table updates a follower's recency only upon insertion. Also, a follower compares its recency only against followers of the same leader.

The simplicity of the cache table structure enables fast lookup, insertion, and eviction. For lookup, the table uses hash maps to locate leaders and followers. For insertion and eviction, the table organizes leaders and followers in each LRU domain using doubly linked lists, where items in a list are ordered by recency. Therefore, the time complexity of lookup, insertion, and eviction operations is $O(1)$.

Standard libraries can readily support the construction of the table thanks to its simple structure. Our prototype implementation simply uses the `OrderedDict` type provided by Python.

The memory consumption of the cache table can be estimated by counting the maximum number of tokens retained by the table, bounded by $O(\text{LL} \cdot \text{LC} \cdot \text{FL} \cdot \text{FC})$. An ordered hash table like `OrderedDict` needs extra memory to maintain metadata including the bucket status and to store pointers in each linked element, but the asymptotic bound remains the same.

3.2 Draft Generation and Validation

Cacheback generates a tree of draft tokens by recursively querying the cache table. Figure 2 shows an example of a draft tree generated for the prefix tokens "At dawn the fox" using a cache table where LL = 2 and FL = 2. We use a word to represent a token for illustrative purposes. Tokens that are not KV-cached are those that were just accepted in the last decoding step. In the first draft generation iteration, Cacheback queries the table with

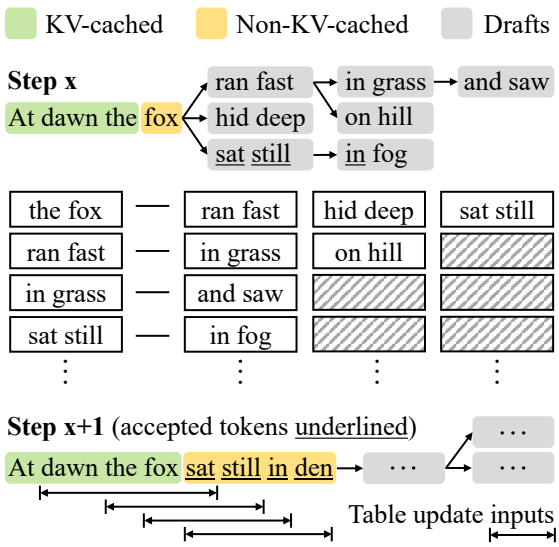

Figure 2: Overview of decoding steps. Cacheback generates a draft tree by recursively querying the cache table and verifies it in one forward pass of the LLM using tree attention. In this example, the last draft branch except its last token is accepted. Cacheback subsequently updates the cache table with the accepted tokens over a sliding window.

the last two tokens in the sequence ("the fox") and receives the list of followers containing "ran fast," "hid deep," and "sat still." In subsequent iterations, Cacheback attempts to grow the tree from its leaf nodes, querying the table with the last LL tokens of the sequence when following the path from the root to a leaf node. The growth of the draft token tree follows a breadth-first-search pattern. Draft generation stops when either none of the leaf nodes has a follower in the cache table or the size of the tree reaches a predefined threshold. We call this threshold the total draft length (TDL), which counts the number of draft tokens plus the number of non-KV-cached tokens.

Cacheback further introduces the chaining-reserved tokens (CRT) parameter to control the width versus depth of the tree. CRT denotes the number of draft tokens reserved for the second or deeper level of the tree. Without setting CRT, a draft tree can become wide enough that its first level exhausts TDL.

Cacheback employs tree attention to efficiently validate the draft tokens in one forward pass of the LLM. Cacheback builds a custom attention mask in which a token attends only to its ancestor tokens in the draft tree. The LLM can then validate all branches of the tree in parallel as one input. A custom GPU kernel may further improve the per-

formance of Cacheback. We leave this as a future direction to explore.

An LLM forward pass always generates one more token in addition to the accepted tokens from the draft. Therefore, a decoding step generates one token in the worst case when it accepts no draft token, or one plus the longest branch length in the best case.

At the end of each decoding step, Cacheback updates the cache table with a sliding window over the accepted tokens, as shown in Figure 2. The window captures all newly observed leader-follower pairs generated by the recent step.

3.3 Table Initialization

Cacheback employs a dual-table approach to improve cold-start performance. In addition to the dynamic cache table just described, Cacheback prepares an additional *frozen* table offline, filling it with frequent leader-follower pairs observed in large training corpora. In each decoding step, Cacheback first queries the dynamic table to form the draft tree and then the frozen one to further grow the tree if TDL still allows. The frozen table disregards insertions during decoding, and only the dynamic table is updated with accepted tokens.

Moreover, at the beginning of each decoding task, Cacheback initializes the dynamic table with a sliding window over the prompt to populate it with the input context.

4 Experiment

We conduct experiments on a desktop machine with an AMD Ryzen 5965WX CPU and four NVIDIA RTX 4090 GPUs. We use one, two, and four GPUs to run the Vicuna 7B, 13B, and 33B models, respectively.

To compare the performance of Cacheback against other training-free model-agnostic methods, we use the SpecBench (Xia et al., 2024) testing framework and dataset. We modify SpecBench in two ways to ensure fairness among evaluated methods. First, for stateful methods, including SAM Decoding (Hu et al., 2024), Token Recycling (Luo et al., 2024), and our approach, we reset the state object before running each test case. Second, we build the static automaton as described in the SAM repository (Hu, 2024) and include it when running SAM on SpecBench. We also fix the SpecBench implementation of Retrieval-based Speculative Decoding (REST) (He et al., 2024) and Token Recycling so that the code can run with multiple GPUs.

For Cacheback, we configure the cache table with LL = 1, LC = 2^{20}, FL = 3, FC = 128, TDL = 96, and CRT = 16. We pick these values empirically for the best performance. We configure LC to be large to reduce the cache-miss rate and FC to be large to saturate TDL. With our configurations, a fully populated table uses at most a few GiB of DRAM, as analyzed in §3.1.

We build the frozen table by randomly sampling 1% of the OpenWebText dataset (Gokaslan et al., 2019). The building procedure first picks the most frequent LC n-grams of length LL as the leaders in the table. Then, for each leader, it selects the most frequent FC n-grams of length FL that appear after the leader in the dataset for inclusion in the follower list.

As shown in Figure 3, despite being simple, Cacheback is on par with SAM Decoding based on suffix automata. Furthermore, Cacheback outperforms Prompt Lookup Decoding (PLD) (Saxena, 2023) that runs brute-force string matching, Lookahead Decoding (Lookahead) (Fu et al., 2024) that employs parallel Jacobi iteration, REST that leverages a database, and Token Recycling that constructs an adjacency matrix to generate drafts. We note that the testing framework currently cannot run the Lookahead method with multiple GPUs. Our results demonstrate that simpler methods can be just as effective as more sophisticated ones.

Notably, the translation task is particularly challenging for all evaluated SD methods. This is partly because the generated words have very little relevance to the input context at the token level. Cacheback's lead in the translation domain demonstrates that our approach can effectively leverage language locality in the output text for speedup, suggesting its rapid adaptation to a new domain and its effectiveness for low-resource languages for which training a draft model is difficult.

We observe that the performance of Cacheback exhibits an interesting pattern across different settings of LL and FL, as shown in Figure 4, which plots the average speedup ratio of Cacheback on SpecBench with Vicuna 7B while varying these two parameters. In this figure, we set LC, FC, TDL, and CRT to the same values as before, but the trend remains the same with other configurations of these parameters. Cacheback consistently achieves the best performance when LL is set to 1 and FL is around 3. It may seem counterintuitive at first that Cacheback runs fastest when LL

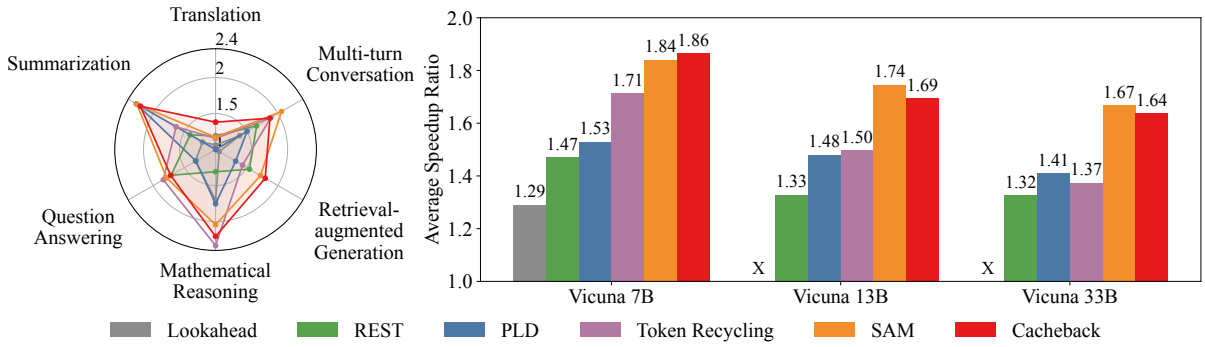

Figure 3: Wall-clock speedup ratio on SpecBench with Vicuna models. The radar plot shows the speedup on different task categories when running Vicuna 7B. Cacheback achieves superior or comparable performance to other training-free model-agnostic methods.

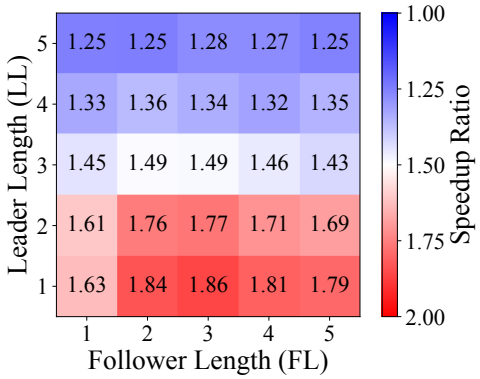

Figure 4: Speedup ratio of Cacheback on SpecBench running Vicuna 7B with different LL and FL settings.

= 1. However, Cacheback's effectiveness is partly attributable to having multiple draft candidates in a tree, which increases the probability that some draft tokens are accepted. With LL = 1, the cache table can return more candidate followers with recent occurrences. Meanwhile, FL = 3 strikes the best balance between the number of drafts and draft length. A greater number of drafts increases the probability that at least some tokens from a draft will be accepted. On the other hand, if a draft is very accurate, increasing draft length will result in more tokens being accepted in a step.

Finally, we demonstrate in Table 1 that the dual-table approach is essential for Cacheback to achieve good performance. Without the frozen table, both the mean accepted tokens (MAT) and the speedup ratio drop significantly due to the cold-start problem. Moreover, since the frozen table cannot reflect the specific context of a decoding loop, using it alone is also suboptimal.

Configuration	Speedup	MAT	Token/s
Dual	1.86×	2.42	103.71
No Frozen	1.64×	1.96	91.32
Only Frozen	1.28×	1.59	68.11

Table 1: Speedup ratio, mean accepted tokens (MAT), and average token generation speed of Cacheback running SpecBench with Vicuna 7B under different table configurations. The dual-table approach is necessary for good performance.

5 Conclusion

We propose Cacheback Decoding, a simple yet effective speculative decoding method that leverages LRU cache tables to exploit the locality in language to accelerate LLM inference. Our results show that Cacheback achieves superior or comparable performance to many sophisticated training-free model-agnostic methods. Due to its simplicity, Cacheback can be easily integrated into existing LLM frameworks. Moreover, because Cacheback organizes draft tokens as a tree, it can be combined with other SD methods for further speedup by inserting their predicted drafts as additional branches, similar to the combination of SAM decoding and EAGLE (Hu et al., 2024). Finally, with Cacheback's inherited strengths from CLMs, our approach exhibits rapid adaptation to specific domains, as evidenced by its strong performance in translation.

Acknowledgments

This work was supported in part by National Science Foundation (NSF) Athena AI Institute under Award 2112562. The authors are grateful for the useful feedback from their reviewers.

Limitations

Our exploration of Cacheback has several remaining areas for investigation. The impact of different corpora for initializing the frozen table remains unstudied, as does performance variation across different GPU architectures and LLM models. Additionally, our evaluation is currently limited to the SpecBench dataset, which may not represent all possible use cases. Moreover, the performance of Cacheback exhibits sensitivity to configuration parameters, with optimal settings likely varying across different tasks, language models, and hardware configurations. A theoretical analysis of these parameters is still lacking, and developing an automatic parameter tuning method would be valuable. We leave these investigations for future work.

References

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. 2024. MEDUSA: Simple LLM inference acceleration framework with multiple decoding heads. In *Proc. Int. Conf. Machine Learning (ICML)*.

Philip R Clarkson and Anthony J Robinson. 1997. Language model adaptation using mixtures and an exponentially decaying cache. In *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. Break the sequential dependency of llm inference using lookahead decoding. In *Proc. Int. Conf. Machine Learning (ICML)*.

Richard Futrell, Kyle Mahowald, and Edward Gibson. 2015. Large-scale evidence of dependency length minimization in 37 languages. *Proc. National Academy of Sciences*.

Aaron Gokaslan, Vanya Cohen, Ellie Pavlick, and Stefanie Tellex. 2019. Openwebtext corpus. `http://Skylion007.github.io/OpenWebTextCorpus`.

Zhenyu He, Zexuan Zhong, Tianle Cai, Jason Lee, and Di He. 2024. REST: Retrieval-based speculative decoding. In *Proc. Conf. North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Yuxuan Hu. 2024. Official implementation of SAM-Decoding: Speculative decoding via suffix automaton. https://github.com/hyx1999/SAM-Decoding.

Yuxuan Hu, Ke Wang, Xiaokang Zhang, Fanjin Zhang, Cuiping Li, Hong Chen, and Jing Zhang. 2024. SAM decoding: Speculative decoding via suffix automaton. *arXiv preprint arXiv:2411.10666*.

Frederick Jelinek, Bernard Merialdo, Salim Roukos, and Martin Strauss. 1991. A dynamic language model for speech recognition. In *Proc. Wrkshp. Speech and Natural Language*.

Roland Kuhn and Renato De Mori. 1990. A cache-based natural language model for speech recognition. *IEEE Trans. Pattern Analysis and Machine Intelligence*.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *Proc. Int. Conf. Machine Learning (ICML)*.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. Eagle: speculative sampling requires rethinking feature uncertainty. In *Proc. Int. Conf. Machine Learning (ICML)*.

Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. 2024. Online speculative decoding. In *Proc. Int. Conf. Machine Learning (ICML)*.

Xianzhen Luo, Yixuan Wang, Qingfu Zhu, Zhiming Zhang, Xuanyu Zhang, Qing Yang, Dongliang Xu, and Wanxiang Che. 2024. Turning trash into treasure: Accelerating inference of large language models with token recycling. *arXiv preprint arXiv:2408.08696*.

Apoorv Saxena. 2023. Prompt lookup decoding. https://github.com/apoorvumang/prompt-lookup-decoding/.

Heming Xia, Tao Ge, Peiyi Wang, Si-Qing Chen, Furu Wei, and Zhifang Sui. 2023. Speculative decoding: Exploiting speculative execution for accelerating seq2seq generation. In *Findings of the Association for Computational Linguistics: EMNLP*.

Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. 2024. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. In *Findings of Association for Computational Linguistics*.

Yao Zhao, Zhitian Xie, Chen Liang, Chenyi Zhuang, and Jinjie Gu. 2024. Lookahead: An inference acceleration framework for large language model with lossless generation accuracy. In *Proc. ACM SIGKDD Conf. Knowledge Discovery and Data Mining (KDD)*.

Yongchao Zhou, Kaifeng Lyu, Ankit Singh Rawat, Aditya Krishna Menon, Afshin Rostamizadeh, Sanjiv Kumar, Jean-François Kagy, and Rishabh Agarwal. 2024. DistillSpec: Improving speculative decoding via knowledge distillation. In *Proc. Int. Conf. Learning Representations (ICLR)*.