Turning Trash into Treasure: Accelerating Inference of Large Language Models with Token Recycling

Anonymous ACL submission

Abstract

Massive parameters of LLMs have made inference latency a fundamental bottleneck. Speculative decoding represents a lossless approach to accelerate inference through a guess-and-verify paradigm. Some methods rely on additional architectures to guess draft tokens, which need extra training before use. Alternatively, retrieval-based training-free techniques build libraries from pre-existing corpora or by n-gram generation. However, they face challenges like large storage requirements, time-consuming retrieval, and limited adaptability. Observing that candidate tokens generated during the decoding process are likely to reoccur in future sequences, we propose Token Recycling. It stores candidate tokens in an adjacency matrix and employs a breadth-first-search (BFS)-like algorithm to construct a draft tree, which is then validated through tree attention. New candidate tokens from the decoding process are then used to update the matrix. Token Recycling requires <2MB of additional storage and achieves approximately 2x speedup across all sizes of LLMs. It significantly outperforms existing train-free methods by 30% and even a widely recognized training method by 25%.

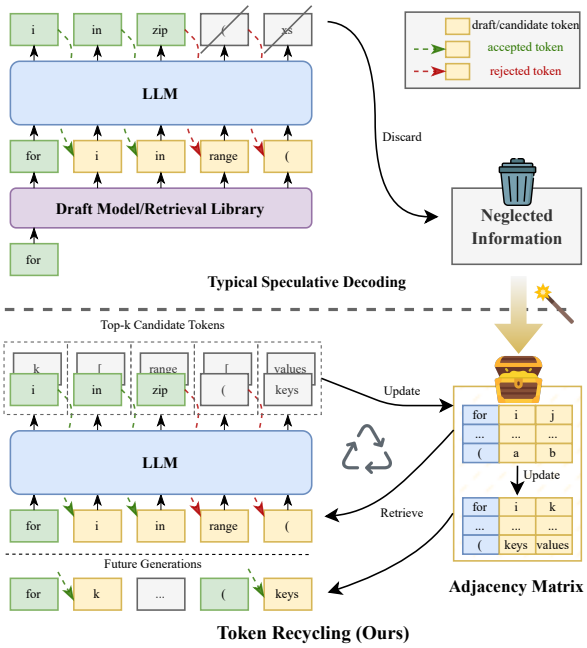

Figure 1: A comparison of typical speculative decoding and Token Recycling (TR). Typical methods draft some tokens and verify them in parallel in one decoding step. Unlike other methods that discard candidate tokens, TR stores them in an adjacency matrix. In future generations, draft tokens are retrieved from the matrix which is updated with new candidate tokens. TR effectively recycles tokens in the decoding process.

1 Introduction

Large Language Models (LLMs) (Brown et al., 2020; Gemini Team et al., 2023; Touvron et al., 2023; Meta, 2024) have becoming the foundation of numerous applications such as chatbots, code assistants, and agents (OpenAI, 2023; Chen et al., 2021; Wang et al., 2024a). However, due to the *auto-regressive* decoding strategy, LLMs can only generate a single token at each decoding step, leading to high inference latency (Brown et al., 2020). The latency mainly comes from transferring billions of parameters from high bandwidth memory to the accelerator cache at each decoding step, rather than arithmetic computations (Kim et al., 2024; Shazeer, 2019; Cai et al., 2024).

Many approaches (Xu et al., 2024; Frantar and Alistarh, 2023; Dao, 2024; DeepSeek-AI, 2024) seek to reduce the latency, with *speculative decoding* as a key lossless technique. This approach employs a *guess and verify* process to obtain multiple tokens during a single decoding step (Chen et al., 2023; Leviathan et al., 2023; Miao et al., 2024; Xia et al., 2023). It first speculates several subsequent draft tokens and then verifies them using the original LLMs. The time cost of verification on multiple tokens is comparable to that of generating one token due to the high parallelism of accelerators. Once some draft tokens are correct, the decoding

steps is significantly shortened without sacrificing quality. To fully utilize the parallelism of accelerators, *tree attention* slightly adjust the attention mask to verify multiple token sequences in one model forward (Cai et al., 2024; Miao et al., 2024).

For effective acceleration, speculative decoding must ensure accurate draft predictions while keeping speculation overhead low. Additional model architectures are constructed to guess the draft tokens, including small draft models (Leviathan et al., 2023; Chen et al., 2023) and parameter-efficient structures (Cai et al., 2024; Lin et al., 2024). However, these approaches require resources for additional training on each LLM. The typical approach to achieve train-free speculative decoding is retrieve-based. In this case, a retrieval library is pre-defined to obtain tokens following the suffix of current content as draft tokens. Several methods have been proposed in this category, each with its trade-offs: **(i)** REST (He et al., 2023) transforms existing corpora into a retrieval library, but *the storage is large, retrieval is time-consuming*, and *the library lacks flexibility* as it's static to any queries. **(ii)** PLD (Saxena, 2023) only retrieves the previous content with minimal cost. However, *it can not predict new tokens or new token combinations*. **(iii)** Lookhead (Fu et al., 2024) construct and update an n-gram library by decoding n times with LLMs. However, *LLMs have to generate n-grams while in inference, causing low efficiency*.

Furthermore, **all speculative decoding approaches fail to fully utilize candidate tokens**, which are multiple possible next tokens generated by LLMs at each decoding step. In greedy decoding, only the top-1 candidate token of accepted tokens is selected as the output, while other candidate tokens, including all candidate tokens from rejected tokens, are discarded, such as 'k' and 'keys' in Figure 1. However, we observe that **when current input tokens reappear in future generations, the following tokens could be candidate tokens generated several steps prior**. Based on the observation, we propose **Token Recycling (TR)**, which utilizes candidate tokens as draft tokens. It stores candidate tokens in an adjacency matrix. Before each decoding step, a BFS-like approach retrieves a draft tree from the matrix, which is then verified using tree attention. Once verified, the newly generated candidate tokens update the matrix. **(i)** The matrix provides a **flexible retrieval library** that is tailored to each query and offers **low retrieval costs** due to its **small size (<2MB)**. **(ii)** Compared to using the previous content solely, candidate tokens naturally include more tokens, providing **many possible continuations**. **(iii)** The construction and update of our library (matrix) utilize the 'trash' tokens **without requiring any additional generation**.

We conduct comprehensive experiments on general benchmark SpecBench (Xia et al., 2024), and specialized dataset on code domain, MBPP (Chen et al., 2021) with Vicuna (Zheng et al., 2023) and Code Llama (Roziere et al., 2023) . The results show that TR greatly exceeds previous train-free approaches, and improves more than 30% on all sizes (7b, 13b, 33b/34b). The speed-up ratio even exceeds the widely used training approach–Medusa, demonstrating its high efficiency.

Our contributions are summarized below:

- A plug-and-play speculative decoding method, Token Recycling is proposed. It firstly recognizes the value of 'trash' tokens and converts them into 'treasure' tokens for acceleration.
- TR requires minimal storage space (<2MB) with a low retrieval cost and covers many new tokens. Continuously updating provides a dynamic retrieval space.
- TR achieves approximately 2x speedup on all sizes of LLMs. It achieves a new SOTA with an improvement greater than 31% compared to previous train-free approaches and even exceeding a training approach.

2 Background

In this section, we overview the speculative decoding. We first define auto-regressive (AR) decoding formally, then discuss speculative decoding, focusing on two key strategies: guess-and-verify and tree attention.

2.1 Auto-Regressive Decoding

AR is the default decoding strategy of LLMs. At each step t, LLMs calculate the probability distribution of the next token given the current content $s = (x_0, x_1, \cdots, x_t)$ which $x_i \in \mathcal{V}$:

$$p_{t+1} = P(x|s;\theta).$$

Here $\mathcal{V}$ is the vocabulary and θ denotes LLM parameters. The next token is selected from p_{t+1} based on the sampling method. Followed Kou et al. (2024), we focus on greedy decoding in this paper, where the next token is:

$$x_{t+1} = \text{argmax}\ p_{t+1}.$$

Candidate tokens are the top-k tokens with the highest probabilities

$$(x_{t+1}^0, x_{t+1}^1, \ldots, x_{t+1}^{k-1}) = \text{argtop}k(p_{t+1})$$

where k is the number of candidate tokens, argtop$k(\cdot)$ returns the indices of the top-k highest values in p_{t+1} and $x_{t+1}^0 = x_{t+1}$.

2.2 Speculative Decoding

Guess and Verify Speculative decoding effectively utilizes the parallel capability of accelerators. Given s, it first guesses n subsequent draft tokens $(\tilde{x}_{t+1}, \cdots, \tilde{x}_{t+n})$. The combination $(s, \tilde{x}_{t+1}, \cdots, \tilde{x}_{t+n})$ is then sent to LLMs for **one** forward pass, resulting in:

$$p_{t+1} = P(x \mid s; \theta),$$
$$\tilde{p}_{t+i} = P(x \mid s, \tilde{x}_{t+1}, \ldots, \tilde{x}_{t+i-1}; \theta), i = 2, \ldots, n.$$

p_{t+1} is the same as AR decoding so the ground truth x_{t+1} is determinable. If the draft token $\tilde{x}_{t+1}$ matches x_{t+1}, then $\tilde{p}_{t+2}$ is assumed to identical to p_{t+2}. Thus, the next ground truth is selected: $x_{t+2} = \text{argmax } \tilde{p}_{t+2}$. This verification process continues until the draft token does not match the ground truth, indicated by:

$$x_{t+j} = \text{argmax } \tilde{p}_{t+j} \neq \tilde{x}_{t+j}.$$

Ultimately, j new tokens are confirmed in one forward pass. The time cost of one forward pass with $(s, \tilde{x}_{t+1}, \cdots, \tilde{x}_{t+n})$ is nearly the same as with s due to the high parallel performance of accelerators. Figure 1 shows an example. The draft tokens are ['i', 'in', 'range', '('] and the output tokens are ['i', 'in', 'zip', '(', 'xs'] after the forward pass. Though 'zip' fails to match 'range', three tokens ['i', 'in', 'zip'] are confirmed in one forward pass.

Tree Attention Traditional causal attention masks are designed for linear sequences, which restricts speculative decoding to verifying one sequence at a time. However, as the sequence lengthens during draft token generation, the number of potential continuations increases. For example, in the draft tree in Figure 2, the token following 'guest' could be 'speaker' or 'speak'. Tree attention modifies the attention mask to verify multiple draft sequences simultaneously. It compresses multiple sequences into a single merged sequence, such as ['guest', 'speaker', 'speak'], while preserving the tree structure through tree attention mask. Each child node attends only to its parent nodes, preventing sibling tokens from interfering with each other. After the LLM processes the merged sequence, all possible sequences such as 'guest speaker' and 'guest speak', along with their corresponding output tokens are extracted based on the tree structure and verified in parallel. The longest correct sequence is selected as the final output. In rare cases, when tokens have identical probabilities, tree attention and AR decoding may select different tokens, but this affects the response quality minimally. The detailed explanation is in Appendix A.1.

In summary, speculative decoding, through *guess and verify* and *tree attention*, improves the inference latency robustly and efficiently.

3 Methodology

Figure 2 provides an overview of Token Recycling (TR). It leverages a hot-start adjacency matrix to store candidate tokens and employs a BFS-like algorithm to construct a draft tree. It utilizes tree attention to verify draft sequences and continuously updates the matrix with new candidate tokens generated during the decoding process.

3.1 Adjacency Matrix Initialization

The adjacency matrix $\mathcal{M}$ is a key component in TR, used to store top-k candidate tokens for each token in the vocabulary:

$$\mathcal{M} \in \mathcal{V}^{|\mathcal{V}| \times k}$$

where k is a user-defined hyperparameter. Each element $\mathcal{M}[i, j]$ indicates that the token $V_{M[i,j]}$ is the j-th candidate token associated with V_i. The use of matrix format, as opposed to other structures like tries, enables efficient parallel processing of candidate tokens, which is crucial for reducing retrieval and update times.

Initially, all elements are set to zero, meaning that a token must appear in draft tokens before it has valid candidate tokens. This initialization leads to the matrix starting with limited predictive capability, potentially causing inefficiencies during the early stages of inference. To mitigate this limitation, we implement a *hot start* strategy. This involves continuing to use the existing matrix, thereby leveraging prior knowledge. Even if queries differ in the domain, candidate tokens often include common expressions and patterns that frequently appear across various queries. Consequently, *hot start* ensures that the matrix has a

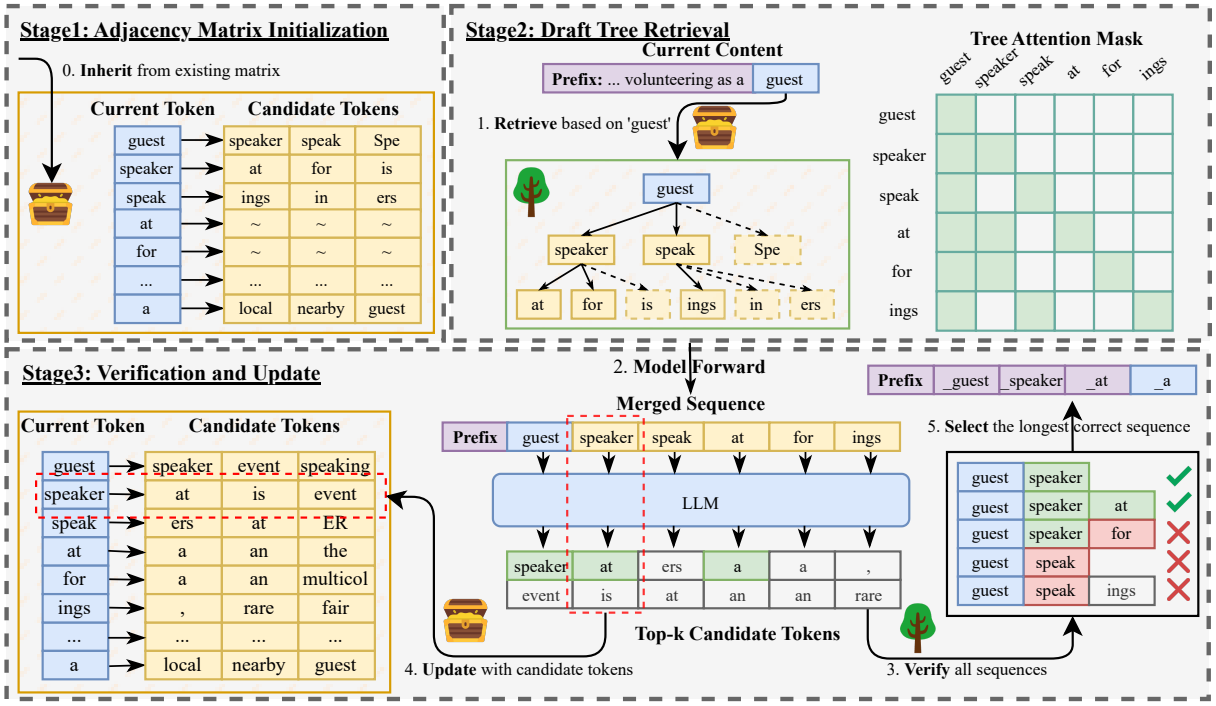

Figure 2: An overview of Token Recycling (TR). The adjacency matrix, initialized by the existing matrix, stores candidate tokens. TR first retrieves a draft tree from the matrix which is then verified through tree attention. After add the longest correct sequence to the content, the new top-k candidate tokens update the matrix.

broader starting point, covering a wide range of potential continuations.

3.2 Draft Tree Retrieval

The adjacency matrix $\mathcal{M}$ stores candidate tokens, which can be used as draft tokens when their corresponding tokens appear later. Directly using the matrix could only determine the immediate next token, such as finding 'speaker' following 'guest' (see Figure 2). Even if 'speaker' is correct, it only slightly improves upon AR decoding, adding just one additional token. In fact, the matrix also holds possible continuations for these candidate tokens, suggesting subsequent tokens like 'at' following 'speaker'. Extending the sequence step by step allows for longer draft sequences. Furthermore, by storing top-k candidate tokens, multiple potential continuations can be explored in parallel for each token, such as 'at' and 'for' following 'speaker'. This BFS process enables the construction of a draft tree with only the adjacency matrix, which can be directly applied to tree attention.

Unlike a complete BFS, we use heuristic rules to define a static and imbalanced tree structure. This tree structure and its construction process are detailed in the Appendix A.2. **Static**: The number of children for each node remains constant across all decoding steps, which facilitates preprocessing and enables efficient parallel operations during layer traversal. Avoiding the need to traverse each node individually significantly reduces retrieval time. **Imbalance**: Nodes positioned earlier in each layer have more children and extend deeper. This allocates computational resources to the most probable continuations since candidate tokens are ordered by probabilities in the matrix.

The BFS-like approach for retrieving the draft tree begins with the matrix $\mathcal{M}$ and the tree structure $Tree$. The root is the last token of current content, like 'guest' in Figure 2. As the root forms the first layer, all candidate tokens for 'guest' are extracted from $\mathcal{M}$, resulting in ['speaker', 'speak', 'Spe']. According to $Tree$, the first layer allows each token to have two children, Therefore, 'speaker' and 'speak', which have the top-2 probabilities, are added to the second layer. The process then proceeds to expand a new layer. All candidate tokens of the second layer are retrieved in parallel, resulting in ['at', 'for', 'is'] and ['ings', 'in', 'ers']. $Tree$ specifies that the first node ('speaker') can have two children, while the subsequent node ('speak') can only have one child. Consequently, the new layer tokens are ['at', 'for'], and ['ings']. This process repeats until the specified depth is reached. The

detailed Algorithm 1 is provided in Appendix A.2.

This retrieval method constructs a draft tree effectively and efficiently with the desired length and variety, which can later be verified by tree attention.

3.3 Verification and Update

The verification of the draft tree aligns with Section 2.2. Merged sequence S is constructed through traversing the draft tree by layers. All potential draft sequences are then verified and the longest correct sequence is selected.

Following verification, the adjacency matrix $\mathcal{M}$ is updated in parallel based on the output distributions $\tilde{p}_{i+1}$ of each draft token $x_i \in S$:

$$\mathcal{M}[\tilde{x}_i] = \text{argtop}k(\tilde{p}_{i+1}).$$

Since multiple preceding tokens may have the same candidate token, duplicates may appear in S, and their output distributions are likely to differ. When performing updates in parallel, CUDA operations may merge these updates, leading to variations in the final result. For example, if x_i appears twice and has two different top-2 output tokens, $[y_0, y_1], [z_0, z_1]$, then $\mathcal{M}[x_i]$ could be updated to exactly one of the following results: $[y_0, z_1]$, $[y_0, y_1]$, $[z_0, z_1]$ or $[z_0, y_1]$. We do not resolve this merging, as adding controls reduces overall performance, as discussed later in Section 5.2.

The update process directly overwrites the previous candidate tokens and leverages the new ones as draft tokens for subsequent decoding steps. This allows the retrieval space to dynamically adapt to the current content, focusing on the most relevant and probable continuations. It also eliminates the necessity for extra operations beyond the standard decoding to update the retrieval space.

In summary, TR capitalizes on the 'trash' present in speculative decoding by implementing a cycling process between candidate and draft tokens. It accelerates inference without the need for additional model structures or training, making it highly adaptable and seamlessly integrated with any architecture or model size.

4 Experiment

4.1 Experimental Setup

Align with previous work (Kou et al., 2024), we focus on common computational redundancy scenarios, specifically greedy decoding with a batch size of one. The following evaluation metrics are used: **Mean Accepted Token (MAT)** (Xia et al., 2024) represents the average number of tokens confirmed in a single decoding step; **Tokens per Second (Ts/s)** measures the number of tokens processed per second; **Speed**up ratio compares the performance relative to HuggingFace's implementation of AR decoding. We set $k = 8$ for $\mathcal{M}$ (<2MB storage in sum) and the draft tree structure is shown in Appendix A.2. All experiments are conducted using Pytorch 2.3 with a single A100-80GB GPU and 128 CPUs under CUDA 12.2.

Datasets and LLMs We conduct experiments on SpecBench (Xia et al., 2024) and MBPP (Austin et al., 2021). SpecBench is a comprehensive benchmark encompassing diverse scenarios including Multi-turn Conversation (MT), Translation (Trans), Summarization (Sum), Question Answering (QA), Mathematical Reasoning (Math), and Retrieval-Augmented Generation (RAG). MBPP is a widely used dataset in code generation, which has a growing demand for efficient generation. These datasets enable a comparative analysis with prior work across both general and specialized domains. We follow the standard practice of utilizing Vicuna (Chiang et al., 2023) for SpecBench and Code Llama (Roziere et al., 2023) for MBPP across three different scales: 7B, 13B, and 33B[1].

Baseline We compare TR with three train-free retrieval-based methods. **Lookahead (Lade)** constructs an n-gram retrieval library through additional n-gram generation during decoding, consuming significant computational resources. **PLD** treats previous content as the retrieval library, which is constrained and cannot introduce new tokens or new token combinations. **REST** builds the retrieval library from existing training datasets, requiring large storage and considerable retrieval time. The static nature of the library also prevents it from adapting to individual queries. Furthermore, we also include a train-need baseline for border comparison. **Medusa** adds multiple additional LM heads in the final layer to predict draft tokens. We focus on losses Medusa-1 since Medusa-2 is lossy. All baselines use their default hyperparameters.

4.2 Main Results

Table 1 shows the performance of TR compared to other methods. On SpecBench, it achieves more than a 2x speedup on the 7B model, nearly 30% higher than the previous train-free methods. Even

[1]The largest model of Code Llama is 34B, for consistency and convenience in our comparisons, we refer to it as 33B.

#Para	Method	SpecBench									MBPP		
		MT	Trans	Sum	QA	Math	RAG	MAT	Ts/s	Speed	MAT	Ts/s	Speed
7B	AR	1.00	1.00	1.00	1.00	1.00	1.00	1.00	54.30	1.00	1.00	56.15	1.00
	Lade	1.42	1.12	1.21	1.21	1.52	1.13	1.64	69.03	1.27	1.66	79.16	1.41
	PLD	1.53	0.98	**2.36**	1.10	1.50	1.74	1.75	83.30	1.53	1.39	66.65	1.19
	REST	1.37	1.05	1.12	1.42	1.06	1.30	1.84	66.29	1.22	2.08	87.08	1.55
	Medusa	1.90	1.57	1.48	1.58	1.87	1.45	2.31	89.41	1.65	-	-	-
	TR	**2.17**	**1.90**	1.94	**1.95**	**2.40**	**1.78**	**2.70**	**110.06**	**2.03**	**2.93**	**131.20**	**2.34**
13B	AR	1.00	1.00	1.00	1.00	1.00	1.00	1.00	39.41	1.00	1.00	41.31	1.00
	Lade	1.29	1.06	1.16	1.12	1.48	1.09	1.63	47.50	1.21	1.73	56.87	1.38
	PLD	1.45	1.01	**2.10**	1.02	1.55	1.65	1.67	57.01	1.45	1.48	52.20	1.26
	REST	1.51	1.14	1.31	1.50	1.17	1.50	1.82	53.34	1.35	2.05	70.13	1.70
	Medusa	1.94	1.66	1.57	1.62	1.98	1.53	2.39	67.92	1.72	-	-	-
	TR	**1.98**	**1.77**	1.89	**1.75**	**2.21**	**1.73**	**2.72**	**74.57**	**1.89**	**3.08**	**93.42**	**2.26**
33B	AR	1.00	1.00	1.00	1.00	1.00	1.00	1.00	18.44	1.00	1.00	19.44	1.00
	Lade	1.32	1.09	1.20	1.17	1.55	1.14	1.61	23.03	1.25	1.70	29.22	1.50
	PLD	1.43	1.06	**1.94**	1.08	1.55	1.41	1.55	25.89	1.40	1.41	25.89	1.33
	REST	1.63	1.27	1.42	1.61	1.29	1.57	1.81	26.99	1.46	2.10	36.85	1.90
	Medusa	**1.98**	**1.75**	1.63	1.68	2.09	1.61	2.32	33.11	1.80	-	-	-
	TR	1.95	**1.75**	1.92	**1.77**	**2.24**	**1.78**	**2.63**	**35.16**	**1.91**	**3.05**	**45.43**	**2.34**

Table 1: Performance of different methods on SpecBench (Vicuna) and on MBPP (Code Llama) across all parameter sizes. Speed is the displayed metric for categories of SpecBench. MBPP results exclude Medusa as it lacks a Code Llama variant. Medusa involves training while others are training-free. **Bold** represents the highest performance.

Method	Memory (MB)	Speed
Lade	105	1.27
PLD	0	1.53
REST	465	1.22
Medusa	>800	1.65
TR	**1.95**	**2.03**

Table 2: The additional memory costs for all methods. Medusa adds extra LM heads to the model, so the memory usage depends on the hidden size and the precision. 800MB is based on a 7B LLM and fp16 precision.

compared to tuning Medusa, it shows an improvement of almost 25%. For the 13B and 33B models, it consistently provides nearly 2x speedup, maintaining the 30% acceleration advantage. These results demonstrate that TR is the most effective train-free method on SpecBench, offering substantial and consistent speedup across all model sizes.

Notably, TR achieves the best speedup across most sub-tasks as well, except it slightly trails PLD on Sum. This may be due to this task often involves many repetitions of previous content. However, the performance gap between TR and PLD narrows as the model size increases, reaching only a 1% difference with the 33B model. This is due to larger models tending to generate new tokens rather than repeat previous content. In other tasks such as MT, Trans, QA, and Math, TR shows a significant improvement of about 40%~70% for the 7B model. This demonstrates the strong generalization of our method across various scenarios. Although the improvement on RAG is less than 3% for the 7B model, it increases with model size, exceeding 10% for the 33B one. This improvement is consistent with the preference of larger models for new tokens. Compared to the general domain, all methods achieve greater acceleration on the code domain due to its higher content redundancy. TR provides approximately 2.3x speedup across all model scales, achieving the SOTA performance.

Furthermore, performances on Trans show the advantages of our method compared to PLD and REST. While PLD shows negligible speedup (close to 1x) and REST achieves its lowest speedup across tasks, TR consistently delivers over 1.75x speedup across all model sizes. Notably, on the 7B model, PLD results in a slowdown, and REST achieves just 1.05x, whereas TR reaches 1.9x. Trans requires generating new tokens continuously, involving minimal repetition of previous content. Additionally, it is highly context-sensitive, making it challenging to find exact matches from any pre-existing database. These pose challenges for PLD and REST. In contrast, the adaptive and diverse retrieval space of TR leads to superior performance. In addition to Speed, TR achieves the highest MAT across both benchmarks. This is attributed to its shorter retrieval times and the avoidance of additional generations like Lade. This allows for deeper and wider draft trees, enabling more tokens to be

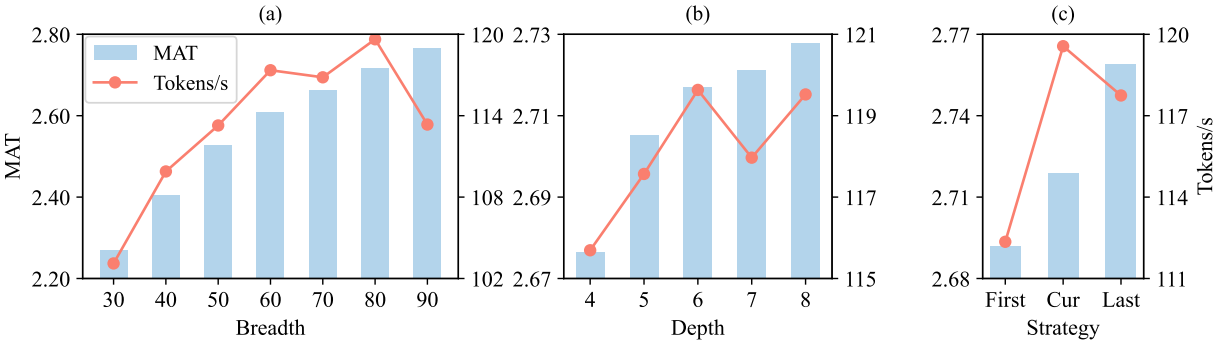

Figure 3: Effects of tree breadth, depth and updating strategies on MAT and Tokens/s are in (a), (b), and (c).

accepted in a single decoding step.

	Tokens/s	Speed
AR	54.98	1.00
Random	95.07	1.73
Zero	102.68	1.87
Fixed	117.43	2.12
Shuffle	118.78	2.16
TR	**119.56**	**2.17**

Table 3: The impact of different initialization strategies of the adjacency matrix. Random means randomly selected from the vocabulary, Zero means all set to zero, Fixed means inherited from a fixed matrix and Shuffle means shuffle the test set.

Table 2 summarizes the GPU memory requirement for all methods. Compared to REST and Lade, TR achieves higher speedup with far less memory. While PLD requires no additional memory, its speedup is limited. Unlike Medusa, our approach is training-free, requires minimal memory, and still achieves superior performance.

TR demonstrates significant improvements across all scenarios, highlighting its efficiency and broad applicability. Importantly, **TR is train-free and self-drafting, allowing for an approximate 2x speedup that can be seamlessly applied as a 'free lunch' to any existing LLM**.

5 Analysis

5.1 Tree Structure

As previously outlined in Section 3.2, our tree structure is static and imbalanced. The tree size is a crucial factor to accelerate. A larger tree allows more tokens confirmed in one decoding step but also introduces more computational overhead, increasing the time required for each decoding step. To investigate the impact of tree size, specifically its depth and breadth, experiments are conducted on MT-Bench using Vicuna-7B.

Breadth Increasing the breadth of the tree allows for covering more possibilities. In Figure 3(a), the breadth is expanded by adding nodes while keeping the depth fixed at six layers. This leads to a consistent improvement in MAT. However, when the breadth exceeds 80, Tokens/s begins to decrease. The additional computational overhead eventually outweighs the benefits of a higher MAT.

Depth Increasing the depth of the tree allows for accepting longer sequences during decoding. In Figure 3(b), with the number of nodes fixed at 80, the depth is gradually increased. MAT initially rises rapidly but eventually shows minimal improvement, while Tokens/s noticeably fluctuates. Because the matrix stores candidate tokens for only adjacent steps, longer sequences weaken the connections between distant tokens. This limitation reduces the effectiveness of increased depth, causing Tokens/s to fluctuate.

5.2 Ablation Study

Hot Start In TR, the adjacency matrix inherits from the previous one. In Table 3, we explore the impact of different initialization strategies. Random means randomly selecting tokens from the vocabulary, while Zero sets all matrix elements to zero. Fixed selects 100 queries from AlpacaEval (Li et al., 2023) (unrelated to the test set), executes them, and stores the resulting matrix. This matrix is then used to initialize each query in the test set. Shuffle refers to shuffling the test set. Compared to the Zero, the irrelevant noise introduced by Random leads to a sharp decrease in performance. Fixed, Shuffle and TR show significant improvements over Zero, suggesting that the prior matrix may capture common patterns that effectively assist subsequent queries. The relatively small difference among them indicates that these patterns are generalizable and not tied to specific tasks or content.

	SpecBench	MBPP
Only Accepted	1.63	1.99
All Draft	**2.69**	**2.93**

Table 4: Mean Accepted Token (MAT) for updating candidate tokens from only accepted or all draft tokens.

Update Strategies Section 3.1 discuss duplicate tokens in the merged sequence during matrix updates. We compare three updating strategies: using candidate tokens from the first occurrence, from the last occurrence, and the current method (merging via parallel CUDA operations). Figure 3(c) indicates that using the last occurrence yields the highest MAT, which may benefit from more contextual information. However, the differences among different strategies in MAT are minimal. In terms of Tokens/s, the current approach significantly outperforms the other two, as it avoids the additional processing required to manage token positions, thereby reducing delays. Speculative decoding is highly sensitive to latency, any extra operation must provide substantial benefits to outweigh its time cost.

Effect of Rejected Tokens During the update, we refresh the candidate tokens for all draft tokens, including both accepted and rejected tokens. To further illustrate the significant effect of trash tokens, we compare two settings: updating only the candidates of accepted tokens versus of all draft tokens. As shown in Table 4, including candidates of rejected tokens significantly improves the MAT. This indicates that rejected tokens also carry valuable information necessary for subsequent decoding. We include a case study in Appendix A.6.

6 Related Work

Efficient inference is crucial for real-time applications and low-resource scenarios. Many strategies have been developed to reduce latency (Zhou et al., 2024b). Among these, speculative decoding (Chen et al., 2023; Leviathan et al., 2023; Miao et al., 2024; Xia et al., 2023) is a losses technique that predicts multiple possible continuations simultaneously. It reduces the number of decoding steps needed without compromising accuracy. Some speculative decoding methods rely on additional draft models to guess draft tokens. These typically involve using smaller models from the same series (Zhao et al., 2024; Spector and Re; Sun et al., 2023; Liu et al., 2024b; Yuan et al., 2024; Gong et al., 2024) or training new models with a shared vocabulary (Leviathan et al., 2023; Chen et al., 2023; Zhou et al., 2024a; Li et al., 2024). It is worth noting that Zhao et al. (2024) also uses rejected tokens but does not include candidate tokens. Additionally, Kou et al. (2024); Wang et al. (2024b) propose training the original LLMs to enable non-aggressive decoding. While effective, these approaches require managing or training multiple models, which can be non-trivial and resource-intensive. Other methods focus on parameter-efficient structures. These approaches minimize the need for complete retraining but still require model-specific training and adaptation, limiting their scalability and general applicability (Lin et al., 2024; Liu et al., 2024a).

Train-free methods construct retrieval libraries to obtain draft tokens (Yang et al., 2023). Lookahead (Fu et al., 2024) generates n-grams through multiple decodings, building a retrieval library that can hit multiple tokens in one step. However, it requires the LLM to generate n-grams while responding to queries, which reduces efficiency. PLD (Saxena, 2023) retrieves only from previous content, resulting in minimal overhead and significant speedup in high-redundancy tasks like summarization. However, it provides little acceleration for tasks requiring the generation of new content, like translation. REST (He et al., 2023) constructs retrieval libraries using existing corpora and performs well in common scenarios. However, this approach requires large storage, time-consuming retrieval, and cannot adapt to each query.

Token Recycling is a train-free, retrieval-based method. It requires no additional generation, covers a broader range of possible continuations, and demands minimal storage with low retrieval costs. The update process ensures the retrieval space remains adaptable.

7 Conclusion

In this work, we introduce Token Recycling, a speculative decoding method for accelerating the inference of LLMs. It utilizes an adjacency matrix to store candidate tokens and retrieve a draft tree, which is then verified with tree attention. The matrix is updated with new candidate tokens generated during decoding. Token Recycling could be integrated seamlessly with existing LLMs and tasks. As a train-free approach, it achieves a speedup of nearly 2x with <2MB of storage, improving over 31% compared to previous train-free approaches.

Limitations

Our study is comprehensive, but has certain limitations that we plan to address in future research. In constructing the draft tree, we use a static tree structure. However, a dynamic tree could be employed instead. While dynamic trees introduce additional complexity, they allow for better adaptation to each decoding step, potentially improving performance by tailoring the tree structure to the specific requirements of each query.

Ethical Considerations

The data for the proposed methods is drawn solely from publicly accessible project resources on reputable websites, ensuring that no sensitive information is included. Moreover, all datasets and baseline models used in our experiments are also available to the public. We have taken care to acknowledge the original authors by properly citing their work.

References

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

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

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. *Preprint*, arXiv:2401.10774.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating Large Language Model Decoding with Speculative Sampling. *Preprint*, arXiv:2302.01318.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *Preprint*, arXiv:2107.03374.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality.

Tri Dao. 2024. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*.

DeepSeek-AI. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *Preprint*, arXiv:2405.04434.

Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive language models can be accurately pruned in one-shot. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10323–10337. PMLR.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. Break the Sequential Dependency of LLM Inference Using Lookahead Decoding. *Preprint*, arXiv:2402.02057.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Zhuocheng Gong, Jiahao Liu, Ziyue Wang, Pengfei Wu, Jingang Wang, Xunliang Cai, Dongyan Zhao, and Rui Yan. 2024. Graph-structured speculative decoding. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 11404–11415.

Zhenyu He, Zexuan Zhong, Tianle Cai, Jason D Lee, and Di He. 2023. Rest: Retrieval-based speculative decoding. *arXiv preprint arXiv:2311.08252*.

Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney,

and Kurt Keutzer. 2024. SqueezeLLM: Dense-and-Sparse Quantization. *Preprint*, arXiv:2306.07629.

Siqi Kou, Lanxiang Hu, Zhezhi He, Zhijie Deng, and Hao Zhang. 2024. CLLMs: Consistency Large Language Models. *Preprint*, arXiv:2403.00835.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast Inference from Transformers via Speculative Decoding. In *Proceedings of the 40th International Conference on Machine Learning*, pages 19274–19286. PMLR.

Xuechen Li, Tianyi Zhang, Yann Dubois, Rohan Taori, Ishaan Gulrajani, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Alpacaeval: An automatic evaluator of instruction-following models. `https://github.com/tatsu-lab/alpaca_eval`.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. Eagle: Speculative sampling requires rethinking feature uncertainty. In *Forty-first International Conference on Machine Learning*.

Feng Lin, Hanling Yi, Hongbin Li, Yifan Yang, Xiaotian Yu, Guangming Lu, and Rong Xiao. 2024. BiTA: Bi-Directional Tuning for Lossless Acceleration in Large Language Models. *Preprint*, arXiv:2401.12522.

Fangcheng Liu, Yehui Tang, Zhenhua Liu, Yunsheng Ni, Kai Han, and Yunhe Wang. 2024a. Kangaroo: Lossless Self-Speculative Decoding via Double Early Exiting. *Preprint*, arXiv:2404.18911.

Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. 2024b. Online speculative decoding. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 31131–31146. PMLR.

Meta. 2024. Introducing meta llama 3: The most capable openly available llm to date.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, volume 3 of *ASPLOS '24*, pages 932–949. Association for Computing Machinery.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Apoorv Saxena. 2023. Prompt lookup decoding.

Noam Shazeer. 2019. Fast Transformer Decoding: One Write-Head is All You Need. *Preprint*, arXiv:1911.02150.

Benjamin Frederick Spector and Christopher Re. Accelerating llm inference with staged speculative decoding. In *Workshop on Efficient Systems for Foundation Models@ ICML2023*.

Ziteng Sun, Ananda Theertha Suresh, Jae Hun Ro, Ahmad Beirami, Himanshu Jain, and Felix Yu. 2023. Spectr: Fast speculative decoding via optimal transport. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024a. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18:186345.

Yixuan Wang, Xianzhen Luo, Fuxuan Wei, Yijun Liu, Qingfu Zhu, Xuanyu Zhang, Qing Yang, Dongliang Xu, and Wanxiang Che. 2024b. Make some noise: Unlocking language model parallel inference capability through noisy training. *arXiv preprint arXiv:2406.17404*.

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 2023*, pages 3909–3925, Singapore. Association for Computational Linguistics.

Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhi-

fang Sui. 2024. Unlocking Efficiency in Large Language Model Inference: A Comprehensive Survey of Speculative Decoding. *Preprint*, arXiv:2401.07851.

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

Yuzhuang Xu, Xu Han, Zonghan Yang, Shuo Wang, Qingfu Zhu, Zhiyuan Liu, Weidong Liu, and Wanxiang Che. 2024. OneBit: Towards Extremely Low-bit Large Language Models. *Preprint*, arXiv:2402.11295.

Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Daxin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023. Inference with reference: Lossless acceleration of large language models. *Preprint*, arXiv:2304.04487.

Hongyi Yuan, Keming Lu, Fei Huang, Zheng Yuan, and Chang Zhou. 2024. Speculative contrastive decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Bangkok, Thailand. Association for Computational Linguistics.

Weilin Zhao, Yuxiang Huang, Xu Han, Chaojun Xiao, Zhiyuan Liu, and Maosong Sun. 2024. Ouroboros: Speculative decoding with large model enhanced drafting. *arXiv preprint arXiv:2402.13720*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. *Preprint*, arXiv:2306.05685.

Yongchao Zhou, Kaifeng Lyu, Ankit Singh Rawat, Aditya Krishna Menon, Afshin Rostamizadeh, Sanjiv Kumar, Jean-François Kagy, and Rishabh Agarwal. 2024a. Distillspec: Improving speculative decoding via knowledge distillation. In *The Twelfth International Conference on Learning Representations*.

Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, et al. 2024b. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294*.

A Appendix

A.1 Identical Probability Tokens

Method	MT-Bench	GSM8K
AR Decoding	6.17	35.2
Tree Attention	6.23	35.2

Table 5: Quality/Accuracy comparison of AR-Decoding and Tree Attention on MT-Bench and GSM8K. MT-Bench results are taken from Cai et al. (2024). It shows that Tree Attention has minimal impact on both answer accuracy and quality.

Floating-point representation in the computer has precision errors, commonly known as 'floating-point rounding errors'. Specifically, the precision of floating-point numbers is determined by the number of bits in the mantissa. In the IEEE 754 standard, the float32 type has a 23-bit mantissa, meaning the smallest representable difference is 2^{-23}, approximately 1.19×10^{-7}. The float16 type, with a 10-bit mantissa, can represent differences as small as 2^{-10}, or about 9.77×10^{-4}. If the difference between two token probabilities is smaller than the precision limit of floating-point representation, these two probabilities will be rounded to the same value, and these tokens will be treated as having identical probabilities during sampling.

AR Decoding uses 'torch.argmax' to return the token with the highest probability. When the probabilities are the same, 'torch.argmax' defaults to returning the one with the smallest index. In Tree Attention, the number of mask tokens is increased compared to AR Decoding, and the attention score of the mask tokens after the softmax operation is not strictly zero, but rather a very small value close to zero. These tiny non-zero values perturb the hidden representations, causing tokens that originally had identical probabilities to now differ slightly, resulting in a different argmax outcome compared to AR Decoding.

Nevertheless, as shown in Table 5, due to the extremely rare occurrence of this issue and the affected probabilities being so close to each other, the impact on experimental accuracy and model performance is negligible.

A.2 Draft Tree Algorithm and Structure

Utilizing tree attention (Miao et al., 2024) to extend the path in the verification phase has become a widely adopted strategy for speculative decoding methods.

Algorithm 1 Static Tree Based BFS

Require: Adjacency matrix $\mathcal{M}$, Static tree structure $Tree$, the last prompt token x_t
Ensure: Merged Sequence S

```
Initialize S ← ∅
Initialize root ← x_t
Initialize the current layer L ← (root)
Initialize the current depth d ← 0
while d < Tree.depth do
    Initialize next layer L_next ← ∅
    Get all candidate tokens of L from M in parallel
    xs = M[L]
    Extract next layer tokens from xs with Tree

    L_next = xs[Tree[d].index]
    Concatenate S and L
    S ← (S; L)
    L ← L_next
end while
return S
```

In Token Recycling, we also use a heuristically constructed token tree to perform the verification. As shown in Figure 5, we construct a static and unbalanced tree inspired by Cai et al. (2024). The number k on a node indicates that it is the k-th candidate token for its parent node. The construction process is below. We begin with a fully balanced 10-branch tree and use an independent validation set to identify the top k nodes that most frequently yield correct tokens. These top k nodes and their children are retained to form a new tree, and the process is repeated to identify the next set of top k nodes. This iterative process continues until performance no longer shows significant improvement. The final tree is determined, and the k is set to consider the maximum number of children across all nodes and the memory requirement. While empirical, this iterative approach has proven to be effective. Further details on tuning the n are provided in Section 5.1. Overall, the tree we construct contains 80 nodes (including the root node) in 6 layers. This means that each forward requires an additional draft input of 79 tokens with a maximum acceptance length of 6.

Building on the tree structure described above, we construct a draft tree for the current content by a BFS-like algorithm in the inference phase. As described in Algorithm 1, we infill the child nodes

of each layer in turn according to the matrix. At last, the merged sequence S is returned and sent to tree attention with $Tree$.

A.3 Time Allocation

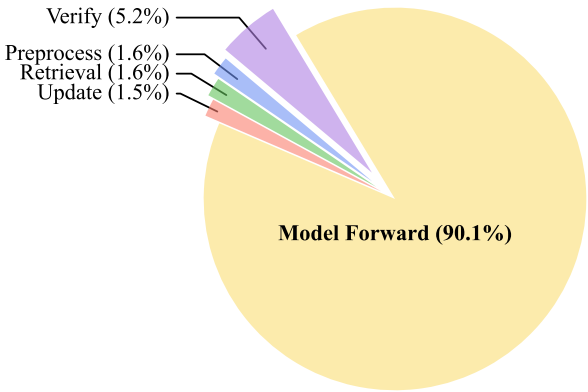

Figure 4: Time allocation for each operation when LLMs respond to a query.

For speculative decoding to be effective, it is essential to maintain a high hit rate while minimizing the time spent on additional operations. We divide each decoding step into several components: *preprocessing* , *retrieving* draft tokens, *model forward* pass, *verifying* draft sequences, and *updating* the matrix, input tokens, and key-value cache. The average time spent on each component is shown in Figure 4. The results indicate that the majority of the time is consumed by the model forward pass. The verification process also takes a significant amount of time due to the need to extract and verify all feasible paths. Retrieving draft tokens and updating operations take roughly the same amount of time.

A.4 Compare With Eagle

TR is Self-Draft and Train-Free, and fair comparisons with other speculative decoding methods that do not require training (e.g., Lookahead, PLD, REST) have already been conducted in section 4.2. However, due to the effectiveness of our approach, our performance significantly exceeds that of these baselines. As a result, we became curious about comparing our method with training-dependent approaches, such as representively, Medusa. The comparison results show that our method even outperforms it a lot. Approaches like Eagle (Li et al., 2024), which rely on additional architectures to generate drafts, may achieve higher performance, but they require collecting datasets for training the small model, and during deployment, they also need extra memory. In contrast, TR only requires <2MB of storage, is train-free, and can be directly applied to all LLMs, achieving nearly a 2x speedup across various model sizes. In Table **??**, we compare the performance of Token Recycling and Eagle on the vicuna-7b-v1.3 model using SpecBench. In comparison, our method uses only 0.39% of the memory required by Eagle, while achieving 88.65% of its speedup performance.

	Memory Cost	Speed Ratio
Eagle	500MB	**2.29**
TR	**<2MB**	2.03

Table 6: Compare Token Recycling with Eagle.

A.5 Reuse Mechanism Analysis

The output of LLM is context-dependent, it makes us curious why reusing candidate tokens from previous generations works. Xiao et al. (2024) analyzed the distribution of attention logits in Transformers and found that the first two layers focus more on 'local' patterns, with recent tokens receiving much more attention. In later layers, the model shifts its focus to the tokens at the beginning of the sequence. To accelerate, candidate tokens need to satisfy both local semantics and long-range dependencies. As shown in Figure 1, draft tokens are divided into accepted and rejected tokens. Accepted tokens are those that appear in the previous query response. In Token Recycling, candidate tokens for all draft tokens are stored, not only accepted ones. Each decoding step involves 79 draft tokens, meaning 79 tokens receive/update their candidate tokens at each step. The quantity of rejected tokens is far more than accepted tokens, and their candidate tokens are also stored in the matrix. In other words, previous generations actually provide a large number of common patterns stored in the matrix, and these patterns often meet local semantic needs.

From two perspectives, the reuse of these common patterns is justified:

For scenarios like sentence transitions, verb collocations, punctuation, or words split into multiple tokens, there is often no need for long-range dependencies. The common patterns stored in the matrix can significantly accelerate the decoding process.

Rejected tokens may not satisfy the long-range dependencies of the previous, but this does not mean they do not meet the long-range dependencies of the current. These tokens may be accepted in

the future. As shown in Table 4, the performance gain from the candidate tokens of rejected tokens is significantly greater than the gain from using only the accepted tokens in the SpecBench experiments. We also include a case study in A.6.

A.6 Case Study

Below, we present a real example from MT-Bench, illustrating how accepted tokens and rejected tokens contribute to the acceleration of Token Recycling. First Round:

- Prefix last token: ['guest']
- Merged sequence with draft tokens: ['guest', 'speaker', '</s>', 'speak', '<0x0A>', 'Spe', 'lect', 'speaking', 'spe', 'at', 'for', '</s>', 'is', 'could', 'can', 'would', '<0x0A>', '<s>', 'sime', 'multicol', 'bolds', '</s>', 'ings', 'in', '<0x0A>', 'a', 'aking', 'ures', 'engag', 'aking', 'a', 'an', 'the', '[', 'our', 'local', '</s>', 'up', 'a', 'an', '</s>', '<s>', 'sime', 'a', 'be', 'help', 'like', '<0x0A>', 'The', 'Home', 'guest', '<s>', '<0x0A>', 'opportunity', '</s>', 'local', 'nearby', 'guest', 'up', 'public', 'guest', '</s>', 'guest', 'public', 'local', 'local', 'The', 'first', 'The', '<0x0A>', 'event', 'community', 'Toast', 'Buddh', 'speaker', 'speaker', 'event', 'time', '.', ',', 'ism']
- Accepted sequence: ['guest', 'speaker', 'at', 'a', 'local', 'event', 'could']

Key observations:

- **Rejected tokens still receive candidate tokens**: For example, ['be'] was not directly accepted, but its candidate tokens were stored.
- **Candidate tokens of accepted tokens are stored**: ['local'] for ['a'] was chosen in this round, but other candidate tokens were also retained.

At this point, the adjacency matrix stores:

- 'be': ['able', 'onto', 'mistaken', 'wrong', 'the', 'interested', 'persu', 'a']
- 'a': ['local', 'time', 'personal', 'professional', 'few', 'unique', 'great', 'low']

Second Round:

- Current prefix last token: ['could']
- Merged sequence with draft tokens: ['could', 'be', 'provide', '</s>', 'help', 'offer', 'present', 'not', 'actually', 'a', 'an', 'the', '</s>', 'just', 'one', 'benef', 'both', ',', "'", 'you', 'for', '<s>', 'sime', 'multicol', 'you', 'overcome', 'some', '</s>', 'public', '<unk>', 'great', 'fant', 'wonderful', 'valuable', 'unique', 'perfect', '</s>', 'ter', 'up', 'event', 'local', 'up', 'local', '<s>', 'not', 'of', 'ited', 'models', 'such', 'like', 'my', 'The', 'as', 'comp', '<s>', 'opportunity', 'guest', '</s>', 'speaker', 'way', 'astic', 'asy', 'bl', 'ins', 'guest', 'coming', 'coming', 'as', 'first', 'a', 'at', 'is', 'would', 'event', '<s>', 'natural', 'public', 'a', 'a', 'an', 'could']
- Accepted sequence: ['could', 'be', 'a', 'great']

At this point:

- **Candidate tokens from rejected tokens are matched**: 'be' correctly predicted 'a'.
- **Candidate tokens from accepted tokens are matched**: 'a' successfully predicted 'great'.

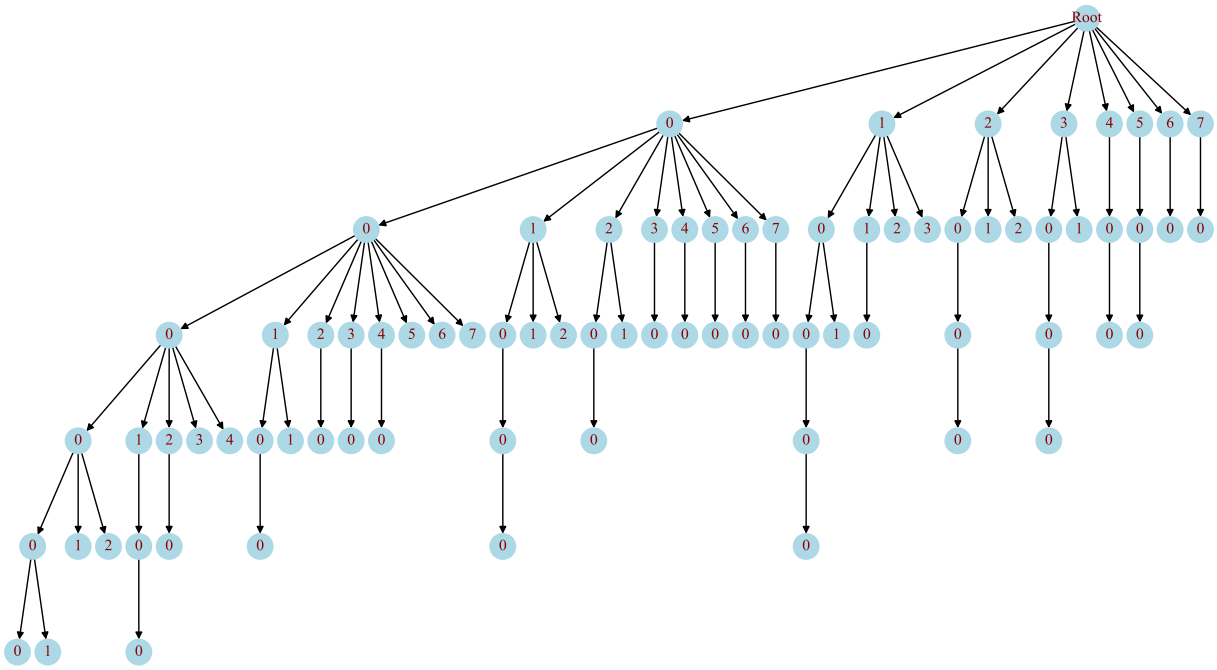

Figure 5: The static tree used in Token Recycling.