# Snakes and Ladders: Accelerating State Space Model Inference with Speculative Decoding

**Yangchao Wu**[1*] **Yonatan Dukler**[2†] **Matthew Trager**[2] **Alessandro Achille**[2]

**Wei Xia**[2] **Stefano Soatto**[2]

[1]**UCLA** [2]**AWS AI**

## Abstract

Speculative decoding is a method for accelerating inference in large language models (LLMs) by predicting multiple tokens using a smaller 'draft model' and validating them against the larger 'base model.' If a draft token is inconsistent with what the base model would have generated, speculative decoding 'backtracks' to the last consistent token before resuming generation. This is straightforward in autoregressive Transformer architectures since their state is a sliding window of past tokens. However, their baseline inference complexity is quadratic in the number of input tokens. State Space Models (SSMs) have linear inference complexity, but they maintain a separate Markov state that makes backtracking non-trivial. We propose two methods to perform speculative decoding in SSMs: "Joint Attainment and Advancement" and "Activation Replay." Both methods utilize idle computational resources to speculate and verify multiple tokens, allowing us to produce 6 tokens for 1.47× the cost of one, corresponding to an average 1.82× wall-clock speed-up on three different benchmarks using a simple *n*-gram for drafting. Furthermore, as model size increases, relative overhead of speculation and verification decreases: Scaling from 1.3B parameters to 13B reduces relative overhead from 1.98× to 1.22×. Unlike Transformers, speculative decoding in SSMs can be easily applied to batches of sequences, allowing dynamic allocation of resources to fill gaps in compute utilization and thereby improving efficiency and throughput with variable inference traffic.

## 1 Introduction

Autoregressive large language models (LLMs) require a costly model call to generate each new token, by processing a long sliding window of input tokens. Several methods have been proposed to reduce the cost of such 'decoding' process; in particular, *speculative decoding* [23, 27, 14] consists of 'drafting' multiple tokens using a simpler 'draft model,' and then 'verifying' their consistency with the original 'base model.' When verification fails, speculative decoding backtracks to the most recent valid token to resume generation. Since drafting and verification introduce overhead cost, the effectiveness of speculative decoding hinges on leveraging computational resources under-utilized by the decoding process. Hardware utilization depends on how well the algorithm structure fits the hardware constraints: Typically, the amount of computation needed to produce a token is small while the memory bandwidth needed to load weights and activations hits hardware limits. This opens a

[*]Work conducted during an internship at AWS

[†]Correspondence to dukler@amazon.com

4th NeurIPS Efficient Natural Language and Speech Processing Workshop (ENLSP-IV 2024).

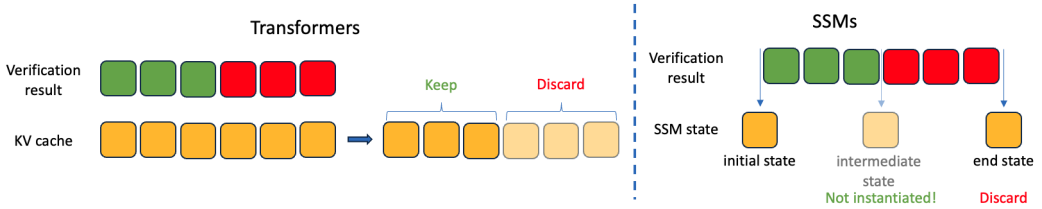


Figure 1: Illustration of the backtracking challenge in applying speculative decoding to State-Space Models (SSMs). Assume a draft model provides 6 tokens for verification, with the green tokens being correct and the red ones incorrect. For transformer models, the key-value caches store all the necessary information to easily backtrack to the state after the last correctly drafted token. However, for SSMs, the required intermediate states are not instantiated in global memory and cannot be easily recovered from the final state. As a result, the correct state must be recomputed.

compute gap that can be filled by verifying multiple tokens simultaneously at minimal cost, thereby increasing arithmetic intensity [26] and decreasing cost with minimal latency.

Unlike Transformers, State Space Models (SSMs) [18, 10] do not re-process past tokens but instead maintain a separate 'Markov state' – a vector which encodes all past tokens for the purpose of generating new ones. Despite reducing inference complexity from quadratic to linear in the input size, inference in SSMs is still costly. Furthermore, the diminished cost of producing a token results in even lower hardware utilization, which makes speculative decoding even more attractive for SSMs since more computational resources are available at essentially no cost. However, once the state of an SSMs is causally updated, the old state is no longer needed and therefore not stored. This makes backtracking not as straightforward as in Transformers, whose state is a sliding window of past tokens (Figure 1).

Our goal in this paper is to enable speculative decoding in SSMs, by introducing methods to leverage under-utilized compute resources to generate multiple tokens at a fraction of the cost of multiple passes through the base model.

### 1.1 Speculative decoding with SSMs

The technical challenge to enable speculative decoding in SSMs is to devise (i) an efficient mechanism to verify multiple tokens in a single forward pass, and (ii) a method to backtrack to the most recent valid state according to (i). For instance, if the first 3 out of 5 draft tokens are considered acceptable by the verification stage, the state should be updated to reflect the first 3 tokens but not the 2 inconsistent tokens. This is straightforward in Transformers: All computation but self-attention is independent for each token, and self-attention can be computed for multiple tokens in a single forward pass due to causal attention masking. Furthermore, the KV-cache, which represents the state of the Transformer [32], is already indexed by the input tokens, so to recover a state one can simply discard the cache beyond the desired index.

In SSMs, (i) requires developing new efficient kernels for multi-step prediction that avoid excessive memory movement, while the 'backtracking problem' (ii) requires novel methods to avoid re-computing the state anew. We propose two: "Activation Replay" stores partial computation from past states that facilitate state backtracking, while "Joint Attainment and Advancement" tracks a desired state and assigns additional computation to predict the next verified state jointly as part of the next forward pass. The details for each of the approaches are described in Section 3.

Since speculative decoding is all about leveraging under-utilized resources, it is important to be able to handle the case where available resources fluctuate. If a speculative decoding method can efficiently generate multiple sequences of varying length, one can modulate the number of sequences, or their length, depending on resources available at any given time. Yet all existing work is focused on accelerating a single sequence at a time. For Transformers, multi-sequence parallel speculative decoding is challenging since the KV-cache grows at different rates for different sequences. In SSMs, on the other hand, the fixed-size state allows generating sequences with different length while maintaining a uniform, parallel batched state. Section 4 shows how to extend our method to multi-sequence generation, and how to adapt the batches to variable inference traffic.

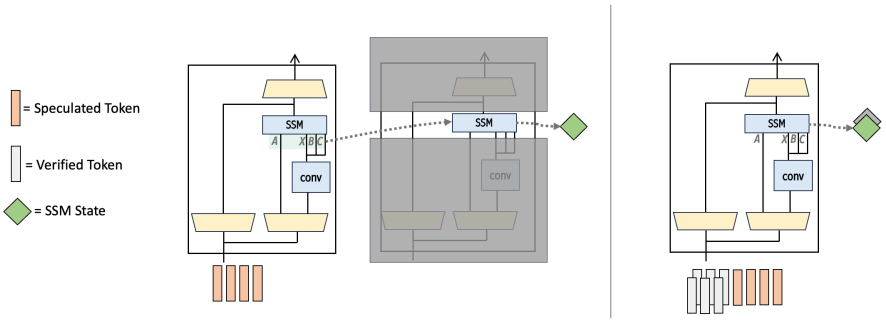


Figure 2: We propose two approaches for speculative decoding of SSMs, here illustrated on a Mamba block [10]. **Left:** *Activation Replay* solves the state backtracking problem by caching activations (the inputs to the SSM block) and re-running only the state update kernel until the last verified token. **Right:** *Joint Attainment and Advancement* evaluates an additional sequence of tokens and recovers the SSM state at the last verified token while performing another verification step.

Our contributions can be summarized as:

1. A new framework for speculative decoding in State Space Models with two methods to overcome the backtracking challenge, with different memory/compute cost profile. Using a Mamba-2 7B[7] SSM as the base model, one of our methods have the potential of generating 6 tokens at $1.18\times$ the cost of generating a single token.
2. End-to-end generation benchmark using low-cost, $n$-gram based drafting that yield a $1.82\times$ wall-clock speed-up on average across MT-Bench [33], GSM8K [6], and HumanEval [5].
3. Empirical analysis with different model sizes, showing favorable scaling in both of our methods thanks to reduced relative overhead for all configurations tested.
4. The first multi-sequence speculation approach for optimizing variable traffic patterns of deployed LLMs via apportioned speculation. We achieve an increase in token throughput for traffic batches ranging from 1-16 sequences up to 29%, as shown in Section 4.

## 2 Related Work

**Speculative decoding** consists of using a more efficient model to generate tokens and the original model to test their consistency [28, 15, 4, 29, 23]. The draft tokens could be generated with independent models [15, 11], fine-tuned [28, 12, 27], by the base model itself using special 'self-drafting' heads [3, 16, 23, 2], or from the model alone [21, 9]. Different inference methods and verification strategies have also been considered, with some works focusing on verifying a single sequence [28], and others using "token-tree verification" [19, 16, 3], which makes use of a special attention mask to verify a branching set of possible strings in parallel. We refer the reader to [29] for a general survey on speculative inference methods.

**State-Space Models** are residual stacking of discrete-time state space models [22] inferred from observed sequences [18]. Specifically, Mamba stacked input-dependent (bilinear) models [13] (referred to as 'selective' in [10]) with simplified (diagonal) state transition matrices which enabled scaling to LLM-size [7, 17, 8]. The backtracking challenge in state space models has been studied extensively in the field of multi-target tracking and data association [1]. For small-scale state space models, KALMANSAC [24] performed speculative decoding by generating multiple trajectories with different random mixes of input data. For large-scale SSMs, after submission of this paper we became aware of concurrent work [25]. To the best of our knowledge, these are the only two existing methods for speculative decoding in SSMs.

## 3 Method

A verification framework requires two key ingredients: 1) an efficient method for validating speculated sequences, by computing the output of multiple tokens with a single model call; and 2) the ability to

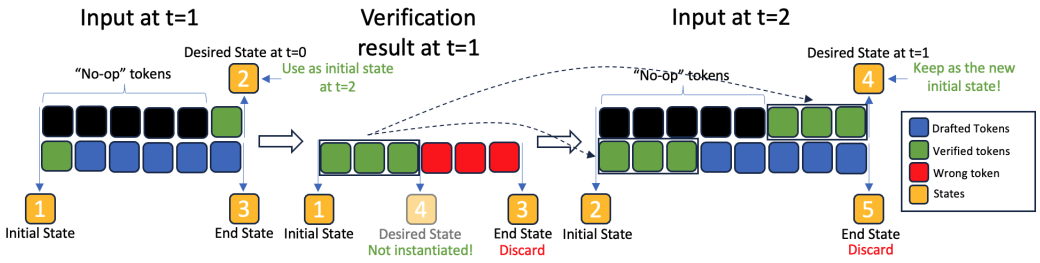


Figure 3: Illustration of Joint Attainment verification process with 1 draft sequence with 5 speculated tokens. The input to the verification model includes an additional sequence of the verified tokens from the previous verification step the recover the final verified state at the previous iteration. Note that the initial state at time $t$ is always the last verified state at time $t-2$.

backtrack to the state corresponding to the last verified token, based on the outcome of the verification stage. The latter is more challenging due to the causal nature of state-space model (SSM) state updates, and is a core focus of our framework. In Section 3.1, we introduce two approaches to address the state backtracking problem. Then, in Section 3.2, we investigate the efficiency of the original Mamba-2 kernels and propose a new optimized kernel to perform parallel verification on short sequences. Finally, in Section 3.3, we present additional implementation details that further optimize the efficiency performance of our framework.

### 3.1 Backtracking the state

Efficient SSM implementations such as [7] typically avoid instantiating intermediate states in GPU memory during the forward pass. This minimizes the memory allocation and bandwidth required for each kernel, outputting only the final state after processing the entire input sequence. This poses challenges for speculative decoding, where a draft sequence may be only partially correct, and the state that yielded the last valid token is not readily available. Since it is prohibitive to store all intermediate states during a multi-step forward pass, we need to backtrack to the last stored state and advance it based on the number of tokens verified to be consistent with the model's output before each forward pass.

**Joint Attainment and Advancement.** One way to recompute the updated state is to include an additional sequence in the input to the next verification pass in order to recompute the state at the last-verified position. Suppose at iteration $t$, we obtain $c < K$ verified tokens out of $K$ drafted ones. Since we don't know the value of $c$ until after the forward pass, we cannot explicitly compute the desired state in the same pass $t$. Therefore, in iteration $t+1$, in addition to the drafted sequences for verification, we include a 'backtracking sequence' as an additional sequence in the input batch. This backtracking sequence is used to generate the state after the $c$ verified tokens from the previous iteration. The other drafted sequences in the same input batch will start with the same $c$ verified tokens, followed by $K$ newly drafted tokens. Once we obtain the updated state using the backtracking sequence, we use this as the initial state for all sequences in the next iteration. See Figure 3 for an illustration of the approach. To perform this efficiently with batch processing, we introduce a "no-op" token that signals no state update. The backtracking sequence is then left-padded with these "no-op" tokens, allowing us to compute the updated state in parallel while continuing the verification. Specifically, we use a mask to set the input-varying state transition matrices $\bar{A}$ to the identity and the input matrices $\bar{B}$ and $C$ to zero when processing "no-op" tokens. We also implement a new kernel that performs the causal convolution operation while accounting for the "no-op" tokens in the backtracking sequence. In summary, in the input of each forward pass, a padded backtracking sequence is used to update the SSM states based on the verification result from the previous forward pass, while the other sequences continue the verification for the current pass. The updated SSM states are then used as the initial state for all sequences in the next forward pass.

**Activation Replay ("ARP").** The Joint Attainment approach requires the full computation of a block (including input projection, casual convolution, SSM state update, and output projection) to be repeated on the verified tokens in the next verification step. To reduce this repeated computation, we

| seqlen | BS=1 | | | BS=4 | | | BS=16 | | |
|---|---|---|---|---|---|---|---|---|---|
| | SSU | CS | Ours | SSU | CS | Ours | SSU | CS | Ours |
| 1 | **0.0074** | 0.064 | **0.0074** | **0.044** | 0.22 | **0.044** | **0.18** | 0.76 | **0.18** |
| 5 | 0.027 | 0.066 | **0.016** | 0.23 | 0.22 | **0.055** | 0.89 | 0.77 | **0.20** |
| 10 | 0.052 | 0.067 | **0.027** | 0.46 | 0.22 | **0.086** | 1.78 | 0.79 | **0.31** |

Table 1: Run-time (in *ms*) of different state update kernels under different batch-size (BS) and multi-token forward sequence length (`seqlen`). SSU stands for Selective-Scan kernel; CS for Chunk-Scan kernel; Ours for Fused Selective-Scan kernel. Since Selective-Scan kernel only takes input of sequence length 1, we use a for-loop to iterate through the sequence length.

| Method | Time (ms) |
|---|---|
| Joint attainment | 34.26 |
| + *Fused-SSU* | 28.17 |
| + *JIT-state-copy* | 27.41 |
| + *both* | **20.74** |

Table 2: Incremental gains (in *ms*) of applying different components of our framework (# of draft = 2, # of spec. = 6). Fused-SSU stands for Fused Selective-Scan-Update kernel.

| | # drafts = 1 | | # drafts = 2 | | # drafts = 3 | | # drafts = 4 | |
|---|---|---|---|---|---|---|---|---|
| # of spec. | ARP | Joint | ARP | Joint | ARP | Joint | ARP | Joint |
| 1 | 19.93 | 16.72 | 20.72 | 17.33 | 21.28 | 17.94 | 21.82 | 19.85 |
| 2 | 20.50 | 17.14 | 21.51 | 18.79 | 22.04 | 19.21 | 22.74 | 20.14 |
| 4 | 21.57 | 17.99 | 22.63 | 19.72 | 25.03 | 21.85 | 24.97 | 22.20 |
| 6 | 22.36 | 18.82 | 23.57 | 20.74 | 25.09 | 22.57 | 26.23 | 24.49 |
| 8 | 22.89 | 19.86 | 24.77 | 21.55 | 26.06 | 24.68 | 30.19 | 25.52 |
| 0 (Mamba2) | 14.10 | 14.10 | 14.10 | 14.10 | 14.10 | 14.10 | 14.10 | 14.10 |

Table 3: Verification framework speed (in ms/decoding step) using our framework with Mamba-2-2.7B with different # of speculations and drafts on 1xA10G Nvidia GPU. Last row is for Mamba-2 auto-regressive generation.

propose Activation Replay. In this method, the values required for recalculating the intermediate state after the last verified state (the inputs for the SSM block) are stored in an activation cache. At the end of an iteration, once we know the number of verified tokens, we can use the cached input activations to recompute the verified state with a separate forward pass focusing only on the state update kernel. Algorithm 2 in the Appendix shows a modified abstracted Mamba-2 block that performs activation replay. Using cached activations avoids the need to recompute dense operations associated with the input and output projection layers. After the Activation Replay forward pass, sequence states are updated to the last verified token in the draft. Then, we use this state to continue verifying new sequences of speculated tokens. With ARP, the input batch does not need to contain a padded verified sequence. The number of steps of the multi-step kernel do not depend on the value of $c$ and instead have a fixed sequence length of $K$. By comparison, in Joint Attainment and Advancement the number of input tokens in time $t+1$ can be in the range $[1+K, 1+2K]$ inclusive, forcing CUDA graph, an optimization technique to be covered in Sec. 3.3, to cache $K+1$ distinct computation graphs for the $K+1$ possible verification outcomes. Moreover, a fixed input sequence length $K$ with no spurious padded sequence is amenable to further kernel optimization since we don't need to handle the padded sequence differently. This also simplifies multi-sequence speculation described in Section 5.3.

**Comparison between the two approaches.** In terms of run-time, Joint Attainment has an edge as the backtracking step is performed in parallel with the verification step while the backtracking step for ARP happens sequentially after the verification step. Meanwhile, in terms of memory footprint, both approaches incur extra memory requirements as compared to auto-regressive generation. Specifically, as we compute the $N$ different drafted sequences in the batch dimension, we need $N$ copies of the states during inference. Joint Attainment requires $N+1$ copies of the states as there is 1 extra padded sequence used for backtracking. ARP, on the other hand, does not rely on an extra copy of the states but requires storing the activations from the last iteration. Since the necessary activations for recomputing the states are much smaller than the states themselves, the Joint Attainment approach actually has a larger memory requirement than ARP, given the same number of sequences and tokens per sequence being verified in one forward pass.

| | | # Params = 1.3B | | # Params = 2.7B | | # Params = 7B | | # Params = 13B | |
|---|---|---|---|---|---|---|---|---|---|
| | # of spec. | ARP | Joint | ARP | Joint | ARP | Joint | ARP | Joint |
| Speed | 6 | 10.23 (1.98×) | 6.84 (1.32×) | 15.39 (1.58×) | 11.82 (1.21×) | 26.66 (1.31×) | 24.04 (1.18×) | 44.07 (1.22×) | 41.99 (1.16x) |
| | 0 | 5.17 | 5.17 | 9.74 | 9.74 | 20.33 | 20.33 | 36.15 | 36.15 |
| Memory | 6 | 3.95 (1.05×) | 4.61 (1.22×) | 6.78 (1.02×) | 7.51 (1.13×) | 14.65 (1.03×) | 15.40 (1.08×) | 26.52 (1.02×) | 27.26 (1.04×) |
| | 0 | 3.76 | 3.76 | 6.60 | 6.60 | 14.21 | 14.21 | 26.06 | 26.06 |

Table 4: Verification framework speed (in ms/decoding step) and memory footprint (in GigaByte (GB)) using our framework with # draft = 1 with Mamba-2 models of different sizes on 1xL40S Nvidia GPU. ARP stands for Activation Replay and Joint stands for Joint Attainment

### 3.2 Efficient multi-token generation

In Mamba-2, there are two implemented modes of state-update computation: auto-regressive mode via the Selective-Scan-Update (SSU) kernel designed for inference, and parallel mode via the Chunk-Scan (CS) kernel designed for training. The Chunk-Scan kernel formulates state update as matrix multiplications which enables efficient performance for updating the states with very long sequences. On the other hand, the SSU kernel can only compute a single update via the state dynamics of Mamba. We benchmark CS and SSU for our settings (predicting a short sequence of new tokens) and observe in Table 1 that the Chunk-Scan kernel is sub-optimal for shorter sequences, while the SSU kernel must be called repeatedly, resulting in unnecessary memory movement from GPU High-Bandwidth-Memory (HBM) to the compute cores. To remedy this, we implement a fused multi-step scan-update kernel that takes a short sequence as input. It iteratively loads the input into the compute cores one step at a time, performs state update using the state dynamics, and stores the output of the state update out to HBM. The kernel is hardware-aware because we avoid storing the large SSM state out to HBM at each step, which reduces the cost of memory transfer. We refer the reader to Sec. A in the Appendix for a detailed algorithm for the kernel. The kernel gives a consistent boost compared to the un-fused SSU and to the CS counterpart for sequence lengths of up to length 10.

### 3.3 Implementation

**Complying with the kernel execution graph** A kernel execution graph such as the CUDA Graph [20] captures the launching of a set of kernels that are part of an underlying computation in order to eliminate their launch overhead. During LLM inference, where each computation is relatively modest, this is paramount for enabling a more efficient utilization of the hardware. For instance, the Mamba-2 implementation utilizes CUDA Graph to improve inference efficiency by more than 3x. While dramatically speeding up inference, a kernel execution graph significantly limits flexibility and the type of operations one can use. First, any computation with a variable sized tensor is forbidden, making the variable advancement steps of speculative decoding difficult to implement. In addition, indexing operations that can use information from the verification stage must be implemented with `IndexSelect` or an Embedding operation. Finally, dynamic control flow during run-time is not allowed. As we implement our framework, we make sure to adhere to these requirements to enable CUDA Graph optimization for high efficiency

**Just-In-Time (JIT) state copying** During regular SSM inference, at the end of each iteration the last SSM state is produced by the kernel and is being stored in a cache. As discussed above, with speculation, we don't know which state to store until after the verification is done. Once we attain the state with either of the two approaches, we can copy it eagerly to update the states for all of the speculative sequences (e.g. in Joint Attainment) to prepare for the next iteration. However it can also be performed Just-In-Time (JIT) before the state needs to be used in the next iteration. We compare the run-time of the two options in Table 2 and find that JIT state copying is significantly faster than the eager approach since the state copying operation with JIT state copying is captured within the kernel execution graph and therefore has a reduced CPU launch overhead.

| Inference strategy | MT-Bench | GSM-8K | HumanEval |
|---|---|---|---|
| Baseline | 70.36 | 70.37 | 70.26 |
| Joint + PNG(6,2) | 112.67 (1.60×) | 137.49 (1.95×) | 133.64 (1.90×) |
| ARP + PNG(6,2) | 103.80 (1.47×) | 122.73 (1.74×) | 120.66 (1.72×) |

Table 5: Inference speed (in tokens/sec.) of our framework using a simple prompt $n$-gram ($n = 6$, $d = 2$ drafts) draft mechanism with Mamba-2 on 1xA10G Nvidia GPU.

## 4 Multi-Sequence Speculation for Dynamic Inference Traffic

In many deployment scenarios, user requests are batched to increase the throughput of generation. In addition, as discussed in the introduction, our framework's flexibility means that we can allocate different speculation budgets in accordance with the volume of requests to the system. For example, if only a single request is processed by the system, there should be ample idle compute for large amount of speculation. On the other extreme, when the system is overwhelmed with a surge of requests, it may be beneficial to completely halt speculation to increase the volume of sequences processed in parallel. While these two extremes are fairly well studied (single-sequence speculation and maximum batch sizes), there is a spectrum of traffic scenarios with different compute utilization and speculation opportunity. Without delving into continuous batching techniques [31] (which currently rely on the transformer architecture), when a user makes an API request for LLM generation, a service has two possible options: immediately process the generation request, or place it in a "periodic-clearing queue" for batching. For the latter, the queue is cleared and sent for processing, either when it reaches a capacity threshold $C_{\text{batch}}$ or after exceeding a waiting threshold $T_{\text{wait}}$. When traffic patterns are very high it is likely that the queue becomes full in the majority of cases and generations are processed with maximal batches. However, for medium and low levels of traffic, the queue will be cleared based on the $T_{\text{wait}}$ to avoid introducing further latency.

Locally, user requests can be modelled by a Poisson distribution with a local traffic volume of $\lambda_{\text{traffic}}$ requests per second. Then for a time threshold $T_{\text{wait}}$, we can model the number of requests to be processed $r$ to be less than $K$ as $P(r(T_{\text{wait}}) \leq K) = \sum_{n=0}^{K} \frac{\exp(\lambda_{\text{traffic}} \cdot T_{\text{wait}})}{n!} (\lambda_{\text{traffic}} \cdot T_{\text{wait}})^n$. In particular when $1 < \lambda_{\text{traffic}} \cdot T_{\text{wait}} < C_{\text{batch}}$, we expect partially filled queues and the opportunity for speculation. For latency requirements, we have that $T_{\text{total}} = T_{\text{processing}} + T_{\text{wait}}$, so $T_{\text{wait}}$ typically has a small margin before regressing total waiting. For our setting, (more in Section 5.3) we prepare multiple execution-graph optimized pipelines with varying levels of speculation to address varying traffic outcomes and seek to optimize throughput by varying the speculation settings for each batch size.

**Extending speculation to multi-sequence generation.** Unlike the sliding window of tokens in Transformers, the state of an SSM encodes the entire history of past tokens with a fixed dimension. This allows the state of different sequences to evolve as a batch, changing the number of tokens while keeping a uniform and parallel batched state representation. This is well-fit to our ARP method that supports evolving the state of different sequences in the batch by different amounts in parallel, followed by uniform multi-step speculation. At the start of the generation, we process the prompt in parallel using the "no-op" token we implemented for Joint Attainment to pad different prompts to the same length. Then, decoding $N$ sequences with $M$ drafts turns into an $N \times M$ batch generation. During model forwarding, we need to cache $N$ states corresponding to each sequence along with the activations for all $N \times M$ speculations. In the replay phase, based on the outcome of the speculation we mask the right activations to replay for each sequence to evolve the states in parallel.

## 5 Experimental Results

Since all generated tokens in our methods are verified against the base model, their effectiveness is measured by improvement in decoding speed relative to that model. We use the open-source Mamba-2 as a reference implementation for ease of reproducibility, and because it is already optimized with custom Triton kernels and CUDA Graph integration [20], thus providing a robust baseline. In Section 5.1 we benchmark inference time of the speculative decoding step under different configurations for the Activation Replay and Joint Attainment approaches, with different ablations. In Section 5.2 we benchmark an end-to-end speculation system by integrating an $n$-gram drafting mechanism within

our framework. We then turn our focus to multi-sequence generation to fill variable compute gaps while increasing throughput in Section 5.3.

### 5.1 Verification framework speed

To benchmark verification speed, we measure the per-step latency, to avoid the confounding factor associated to different drafting methods. Specifically, each speculative decoding step is divided into four phases: (1) drafting; (2) model forward; (3) verification; (4) state copying. We then measure the average run-time of phases (2), (3), and (4) when running generation for 100 tokens on an Nvidia A10G GPU (24GB HBM). The results are shown in Table 3, with 0 corresponding to no-speculation reference run-time. For drafting 2 sequence with 6 speculated tokens with Joint Attainment, we can generate 7 tokens with an 1.47× overhead.

We test our framework across different model sizes (1.3B, 2.7B, 7B, 13B) on an Nvidia L40S GPU (48GB HBM) in Table 4 and observe that the relative overhead of our framework decreases with larger models. With 1 draft and 6 speculated tokens, the overhead drops from 1.98× in a 1.3B model to 1.22× in a 13B model for Activation Replay and from 1.32× to 1.16× for Joint Attainment. Meanwhile, we also recorded the memory footprint of each approach on models of different size. Activation Replay only uses slightly more memory than baseline inference with no speculation, decreasing from 1.09× memory for a 1.3B model to 1.02× for a 13B model. Joint Attainment uses slightly more memory due to the cached copies of the state, but also showed a decreasing trend with increasing model size.

Last, we ran different studies to measure the effect of different features of our methods. In Table 1, we measure the run-time of different kernels given the same input. All numbers are an average of 10 independent trials after warm-up. We find that the Chunk-Scan kernel has a constant run-time at short sequence length, but pays a large overhead as compared to the SSU kernel. Our Fused SSU kernel achieves the same run-time at sequence length 1 but is much faster than both kernels used by Mamba-2 at sequence length of 5 and 10. In Table 2, we ablate the effect of our fused scan kernel as well as JIT state copy. The baseline is the Joint Attainment method with 'eager' state copying and Mamba-2 CS kernel. Running Mamba-2-2.7B on a A10G GPU using a Fused Scan kernel instead of CS improves per-step runtime by 17.8% and using JIT state copy instead of eager state copy improves runtime by 20.0%. Used together, they are reduce runtime by 39.4%.

### 5.2 End-to-end generation

We measure end-to-end generation speed by integrating our framework with a baseline low-cost speculation strategy, the corpus+prompt n-gram drafting method which we refer to as PNG[3] [30]. With PNG and our framework, we can run a number of speculations at a time to increase acceptance rates. To benchmark the system we measure the wall-clock time to generate 100 tokens for each sample in the benchmark using the following common inference benchmarks: MT-Bench [33] (80 samples), GSM8K [6] (test set 1319 samples), HumanEval dataset (162 samples) [5] on one Nvidia A10G GPU. The results are shown in Table 5. We can see that our joint attainment method achieves an average 1.82× speed up across the three benchmarks. With ARP, it achieves an average 1.64× speed up, with an advantage of well formatted input that facilitates multi-sequence generation.

### 5.3 Multi-sequence throughput

For multi-sequence generation, we group sequences in the benchmark datasets into different batch sizes to simulate different levels of traffic and measure the wall-time taken to generate at least 400 tokens for each sequence under different degree of speculation. Throughput is computed by dividing the total number of generated tokens across all sequences by the wall-time and we drop the last batch in the benchmark if it is not divisible by the batch size. We follow the same hardware and setup as the end-to-end generation experiments. Notice that speculative decoding is able to boost the throughput across all input batch sizes, while the gains diminish with larger batch sizes as expected (Figure 4 **Right**). We also observe that in Figure 4 **Left**, with a higher batch size, the benefits brought by having

[3] See for instance `https://github.com/apoorvumang/prompt-lookup-decoding` `https://huggingface.co/blog/dynamic_speculation_lookahead`.

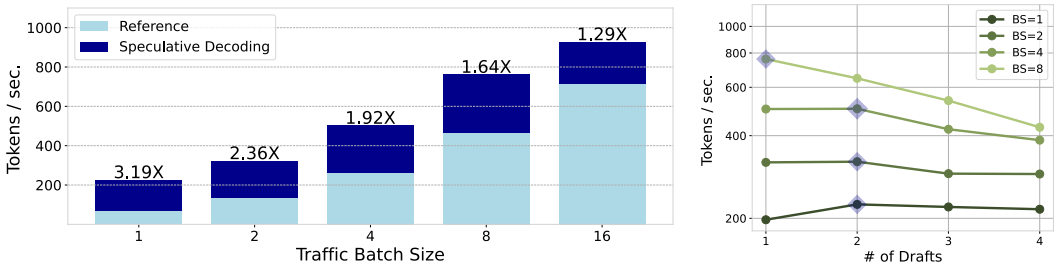


Figure 4: **(Left)**: Relative and absolute improvements in throughput (in tokens/sec.) obtained by applying our framework on a batch of input sequences using the optimal number of drafts. **(Right)**: Throughput (in tokens/sec.) vs. varying number of drafts, for different batch sizes.

multiple speculation drafts for each input sequence are out-weighed by the increase in computation burden starting at BS=8.

## 6 Conclusion

We propose two approaches for speculative decoding in State Space Models with different performance profile: Activation Replay and Joint Attainment, along with an efficient kernel for verifying sequences of short length to enable fast end-to-end speculation. In addition, we also extend our framework to multi-sequence generation to exploit the availability of variable computational resources due to fluctuating inference traffic. In this work, we only tested our framework with a simple $n$-gram drafting methods using the reference implementation of Mamba-2 as a base model. More powerful draft models could easily be combined with our framework.

## References


[1] Yaakov Bar-Shalom, Thomas E Fortmann, and Peter G Cable. Tracking and data association, 1990.

[2] Nikhil Bhendawade, Irina Belousova, Qichen Fu, Henry Mason, Mohammad Rastegari, and Mahyar Najibi. Speculative streaming: Fast llm inference without auxiliary models. *arXiv preprint arXiv:2402.11131*, 2024.

[3] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

[4] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[5] Mark Chen et al. Evaluating Large Language Models Trained on Code, July 2021. arXiv:2107.03374 [cs].

[6] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training Verifiers to Solve Math Word Problems, November 2021. arXiv:2110.14168 [cs].

[7] Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *Proceedings of the 41st International Conference on Machine Learning*, pages 10041–10071, 2024.

[8] Soham De, Samuel L Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, et al. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *arXiv preprint arXiv:2402.19427*, 2024.

[9] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of llm inference using lookahead decoding. *arXiv preprint arXiv:2402.02057*, 2024.

[10] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

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

[12] Sehoon Kim, Karttikeya Mangalam, Suhong Moon, Jitendra Malik, Michael W. Mahoney, Amir Gholami, and Kurt Keutzer. Speculative decoding with big little decoder. In *Neural Information Processing Systems*, 2023.

[13] Arthur J Krener. Bilinear and nonlinear realizations of input-output maps. *SIAM Journal on Control*, 13(4):827–834, 1975.

[14] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[15] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[16] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*, 2024.

[17] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, et al. Jamba: A hybrid transformer-mamba language model. *arXiv preprint arXiv:2403.19887*, 2024.

[18] Anders Lindquist and Giorgio Picci. On the stochastic realization problem. *SIAM Journal on Control and Optimization*, 17(3):365–389, 1979.

[19] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 1(2):4, 2023.

[20] NVIDIA. Cuda 10 features revealed. https://developer.nvidia.com/blog/cuda-10-features-revealed/, September 19 2018.

[21] Andrea Santilli, Silvio Severino, Emilian Postolache, Valentino Maiorca, Michele Mancusi, Riccardo Marin, and Emanuele Rodolà. Accelerating transformer inference for translation via parallel decoding. *arXiv preprint arXiv:2305.10427*, 2023.

[22] Harold W Sorenson. Kalman filtering techniques. In *Advances in control systems*, volume 3, pages 219–292. Elsevier, 1966.

[23] Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. Blockwise parallel decoding for deep autoregressive models. *Advances in Neural Information Processing Systems*, 31, 2018.

[24] Andrea Vedaldi, Hailin Jin, Paolo Favaro, and Stefano Soatto. KALMANSAC: Robust filtering by consensus. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 1, pages 633–640. IEEE, 2005.

[25] Junxiong Wang, Daniele Paliotta, Avner May, Alexander M Rush, and Tri Dao. The mamba in the llama: Distilling and accelerating hybrid models. *arXiv preprint arXiv:2408.15237*, 2024.

[26] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[27] Heming Xia, Tao Ge, Peiyi Wang, Si-Qing Chen, Furu Wei, and Zhifang Sui. Speculative decoding: Exploiting speculative execution for accelerating seq2seq generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3909–3925, 2023.

[28] Heming Xia, Tao Ge, Peiyi Wang, Si-Qing Chen, Furu Wei, and Zhifang Sui. Speculative decoding: Exploiting speculative execution for accelerating seq2seq generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3909–3925, 2023.

[29] Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. *ArXiv*, abs/2401.07851, 2024.

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

[31] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[32] Luca Zancato, Arjun Seshadri, Yonatan Dukler, Aditya Golatkar, Yantao Shen, Benjamin Bowman, Matthew Trager, Alessandro Achille, and Stefano Soatto. B'MOJO: Hybrid state space realizations of foundation models with eidetic and fading memory. *Proc. of NeurIPS*, 2024.

[33] 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. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena, December 2023. arXiv:2306.05685 [cs].

# A Algorithm for Multi-step Selective-Scan-Update kernel

**Algorithm 1** Fused multi-step State-Update kernel. This Algorithm is simplified with the only input being $x, A, B, C, \Delta, h$ in reference to the implementation of [10].

**Notation** $Z^* \leftarrow Z$: load $Z$ into shared memory for computation. $Z \leftarrow Z^*$: Storing Z out to High-Bandwidth-Memory (HBM). Any variable with $*$ means that this variable is in shared memory, everything else is in HBM.

**Shapes** $B$ - Batch, $L$ - Sequence Length, $H$ - Num heads, $P$ - State Dimension , $D$ - Head Dimension, $G$ - Num groups

**Input:** $h_0 : (B, H, D, P)$ - initial state, $x : (B, L, H, D)$ $A : (H, D, P)$, $B : (B, L, G, P)$, $C : (B, L, G, P)$, $\Delta : (B, L, H, D)$

**Output:** $y : (B, L, H, D)$ - Output, $\hat{h} : (B, H, D, P)$ - updated state

1: $h^* \leftarrow h_0$
2: $i \leftarrow 0$
3: **while** $i < L$ **do**
4: $\quad x^* \leftarrow x[:, i, :, :]$
5: $\quad \Delta^* \leftarrow \Delta[:, i, :, :]$
6: $\quad B^* \leftarrow B[:, i, :, :]$
7: $\quad C^* \leftarrow C[:, i, :, :]$
8: $\quad A^* \leftarrow A$
9: $\quad \bar{A}^* \leftarrow \exp(\Delta^* \times A^*)$
10: $\quad \bar{B}^* \leftarrow \Delta^* \times B^*$
11: $\quad h^* \leftarrow h^* \times \bar{A}^* + \bar{B}^* \times x^*$ ▷ State is not stored out to HBM at this point
12: $\quad y^* \leftarrow C^* \times h^*$
13: $\quad y[:, i, :, :] \leftarrow y^*$
14: **end while**
15: $\hat{h} \leftarrow h^*$

# B Algorithm for Activation Replay

Line 1, 3, 5, 8 are performed by the original Mamba-2 block to get the output and update the states. The added operations are the extra $Conv1d\_update$ and $SSM\_update$ at line 2 and 5 to attain the updated states using the correct cached activations. Line 4 and 7 saves the activation of the current iteration for the use of next iteration.

**Algorithm 2** Abstracted Mamba-2 block with Activation Replay. The two functions are simplified abstraction of what is actually used in the implementation.

**Notation** $A[a:b]$ : $a$-th position to $b$-th position of $A$ in the sequence length dimension
**function** $Conv1d\_update(x, B, C, M)$ :
  **return** $(x', B', C', M')$ ▷ Performs 1d convolution on x, B, C, and update conv state M
**function** $SSM\_update(x, B, C, dt, S)$ :
  **return** $(y, S')$ ▷ Performs SSM update with x, B, C, dt to get output $y$ and update SSM state $S$
**Input:** $E$ : token embedding, $S$ : SSM state, $M$ : Conv state, $c$ : number of correct tokens
  $(x_{cache}^{pre\_conv}, B_{cache}^{pre\_conv}, C_{cache}^{pre\_conv})$ : Cached activation for Conv states from last forward pass
  $(x_{cache}^{post\_conv}, B_{cache}^{post\_conv}, dt_{cache})$ : Cached activation for SSM states from last forward pass
**Output:** $\hat{E}$ : New token embedding, $\hat{S}$ : SSM state after the $i$-th correct token, $\hat{M}$ : Conv state after the $i$-th correct token

$x^{pre\_conv}, B^{pre\_conv}, C^{pre\_conv}, dt \leftarrow in\_projection(E)$
$(\_, \_, \_, \hat{M}) \leftarrow Conv1d\_update(x_{cache}^{pre\_conv}[0:c], B_{cache}^{pre\_conv}[0:c], C_{cache}^{pre\_conv}[0:c], M)$ ▷ Getting updated Conv state
$(x^{post\_conv}, B^{post\_conv}, C^{post\_conv}, \_) \leftarrow Conv1d\_update(x^{pre\_conv}, B^{pre\_conv}, C^{pre\_conv}, \hat{M})$ ▷ Actual Conv1d update
$(x_{cache}^{pre\_conv}, B_{cache}^{pre\_conv}, C_{cache}^{pre\_conv}) \leftarrow (x^{pre\_conv}, B^{pre\_conv}, C^{pre\_conv})$ ▷ Saving activations for next forward pass
$(\_, \hat{S}) \leftarrow SSM\_update(x_{cache}^{post\_conv}[0:c], B_{cache}^{post\_conv}[0:c], dt_{cache}[0:c], S)$ ▷ Getting the updated state using the correctly predicted tokens
$(y, \_) \leftarrow SSM\_update(x^{post\_conv}, B^{post\_conv}, C^{post\_conv}, dt, \hat{S})$ ▷ Actual SSM update
$(x_{cache}^{post\_conv}, B_{cache}^{post\_conv}, dt_{cache}) \leftarrow (x^{post\_conv}, B^{post\_conv}, dt)$ ▷ Saving activations for next forward pass
$\hat{E} \leftarrow out\_projection(y)$