Distributed Speculative Inference of Large Language Models

Nadav Timor[W*], Jonathan Mamou[i], Daniel Korat[i], Moshe Berchansky[i], Oren Pereg[i],
Moshe Wasserblat[i], Tomer Galanti[T], Michal Gordon[W], and David Harel[W]

[W]: Weizmann Institute of Science
[i]: Intel Labs
[T]: Texas A&M University
[*]nadav.timor@weizmann.ac.il

Abstract

Accelerating the inference of large language models (LLMs) is an important challenge in artificial intelligence. This paper introduces *distributed speculative inference (DSI)*, a novel distributed inference algorithm that is provably faster than speculative inference (SI) [Leviathan et al., 2023, Chen et al., 2023, Miao et al., 2023] and traditional autoregressive inference (non-SI). Like other SI algorithms, DSI works on frozen LLMs, requiring no training or architectural modifications, and it preserves the target distribution. Prior studies on SI have demonstrated empirical speedups (compared to non-SI) but require a fast and accurate drafter LLM. In practice, off-the-shelf LLMs often do not have matching drafters that are sufficiently fast and accurate. We show a gap: SI gets slower than non-SI when using slower or less accurate drafters. We close this gap by proving that DSI is faster than both SI and non-SI—given any drafters. By orchestrating multiple instances of the target and drafters, DSI is not only faster than SI but also supports LLMs that cannot be accelerated with SI. Our simulations show speedups of off-the-shelf LLMs in realistic settings: DSI is 1.29-1.92x faster than SI. Our code is open-sourced: github.com/keyboardAnt/Distributed-Speculative-Inference

.

1 Introduction

Generative LLMs, such as GPT-4 [OpenAI, 2023], have demonstrated unprecedented results in various applications [Andreas, 2022, Li et al., 2023, Bubeck et al., 2023, Wei et al., 2022]. Despite their potential, the inference latency of LLMs presents a significant challenge and a bottleneck for adoption in real-time applications. For example, in algorithmic trading, the model needs to make rapid predictions to execute trades in milliseconds, and in autonomous driving, the model must act quickly to ensure the vehicle's reliability. This challenge is compounded by existing inference algorithms that do not fully utilize the computational resources that modern hardware offers.

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

Given their usefulness, speeding up the inference of LLMs is an important area of research. Existing efforts to reduce the inference latency can be classified into two main categories: algorithmic innovations and system optimizations. Algorithmic innovations include compressing LLMs through pruning (e.g., [Frantar and Alistarh, 2023, Sun et al., 2024a, Ma et al., 2023]), knowledge distillation (e.g., Hinton et al. [2015], Gu et al. [2024]), quantization (e.g., [Hubara et al., 2018, Frantar et al., 2022, Lin et al., 2024]), and low-rank factorization (e.g., [Hsu et al., 2022, Xu et al., 2023]). On the system side, enhancements such as kernel optimizations [Dao et al., 2022], tensor parallelism [Shoeybi et al., 2019], and low-bit quantization [Yao et al., 2024, Dettmers et al., 2022] are utilized to increase computation speed and reduce operational overhead, directly lowering latency.

Despite reducing the inference time, these methods have a significant drawback: they typically degrade the output quality. Consequently, other approaches acknowledge that some inputs require a very large model, while others can be effectively approximated by more efficient models. The goal of these adaptive methods (e.g., [Elbayad et al., 2020, Bapna et al., 2020, Han et al., 2022, Schuster et al., 2021]) is to channel fewer computational resources for easier inference steps. While many of these solutions are useful in practice, they often require modifications to the model architecture, changes to the training procedure and re-training of the models, without guaranteeing identical outputs.

A recent line of work [Stern et al., 2018] for accelerating the inference of LLMs is based on speculative inference. The idea is to use speculative execution [Burton, 1985, Hennessy and Patterson, 2012] to predict possible continuations of the input prompt using faster *drafter* LLMs that approximate the target LLM, then verify the correctness of the predicted continuations simultaneously by utilizing the concurrency of CUDA-based processors (i.e., *batching*). They provided empirical evidence that their proposed draft-then-verify approach speeds up the inference. Since the introduction of speculative inference [Stern et al., 2018], various papers Leviathan et al. [2023], Chen et al. [2023] have improved this method by introducing novel lossless methods to verify the correctness of token sequences that were generated by the drafter LLMs. Empirically, these approaches lead to speedups in decoding LLMs in practical use cases, such as 2-3x speedups in decoding LLMs of 11B and 70B parameters in some settings. Following this line of work, Miao et al. [2023] extended the verification algorithm of Leviathan et al. [2023], Chen et al. [2023] and showed that their method increases the probability of accepting draft tokens, and proved its losslessness. Following the success of this approach, research in this area has expanded in various directions [Mamou et al., 2024, Li et al., 2024, Cai et al., 2024, Sun et al., 2024b, Zhou et al., 2024, Liu et al., 2023, Joao Gante, 2023].

While traditional methods for SI show how to accelerate the inference time of LMs, they do not take advantage of the possibility of having multiple processing units (e.g., GPUs). In addition, empirical evidence indicates that acceleration happens only when the drafter is very accurate and is significantly faster than the target model. Two key questions then are: (i) *can we reduce the inference time of LLMs by taking advantage of multiple processors simultaneously?* (ii) *can we accelerate the inference time using drafters that are not necessarily very fast or accurate?*

Contributions. In this paper we make the following contributions: 1. We design the first distributed algorithm (across multiple GPUs) for speculative inference of large language models. This algorithm is provably faster than both non-SI and SI methods. 2. We empirically validate, across a wide range of experiments, that our method can speed up the inference time compared to SI, even when fixing the number of processors. 3. We demonstrate that SI requires a drafter model that is both faster and more accurate than the target model. Conversely, our method accelerates inference time even with drafter models that are slower and less accurate ($> 10\%$ latency compared to the target model).

2 Preliminaries

We begin by describing autoregressive language models, next-token prediction, speculative inference and how to measure latency.

Autoregressive language models (LMs) are deterministic, real-valued multivariate functions. An input to an LM is a sequence of vectors. We call these vectors *tokens*, and the sequence a *prompt*. Tokens have a pre-defined dimension, denoted by n_{vocab}. LMs output a vector of real numbers of dimension n_{vocab}, also known as the *logits*. Since prompts to the same LM may vary in length, we simplify the notation of the*forward pass* as follows: $f : \mathbb{R}^{* \times n_{\text{vocab}}} \to \mathbb{R}^{n_{\text{vocab}}}$.

Self-Attention LMs are LMs with pre-defined context length n_{ctx} [Vaswani et al., 2017]. Hence, we represent the forward pass of such LMs in the following manner: $f : \mathbb{R}^{n_{\text{ctx}} \times n_{\text{vocab}}} \to \mathbb{R}^{n_{\text{vocab}}}$. For example, GPT-2 and GPT-3 are Transformers with $n_{\text{vocab}} = 50257$, and context lengths $n_{\text{ctx}} = 1024$ and $n_{\text{ctx}} = 2048$, respectively [Radford et al., 2019, Brown et al., 2020]. In this paper, all LMs are Self-Attention ones with pre-set (frozen) parameters.

We extend the prompt notation such that prompts can have length $l \leq n_{\text{ctx}}$. Self-Attention LMs handle prompts of length $l < n_{\text{ctx}}$ by starting the input sequence with a prefix of $n_{\text{ctx}} - l$ tokens, followed by the l given tokens. LMs ignore the prefix, either by zeroing (masking) the Attention parts corresponding to the prefix or by left-padding with dedicated tokens. In this paper, prompts of length $l < n_{\text{ctx}}$ are the non-masked, non-padded suffix of the input sequence of length n_{ctx}.

Generating the next token is the primary application of autoregressive LMs. This process consists of two steps: computing the forward pass of the LM and then selecting the next token based on the output. The selection can be deterministic or non-deterministic.

Non-deterministic selection procedures apply the softmax function after the forward pass of LMs and sample from the resulting probability vector:

$$\text{softmax} : \mathbb{R}^{n_{\text{ctx}} \times n_{\text{vocab}}} \to [0, 1]^{n_{\text{vocab}}} \text{ such that the entries sum to 1.} \tag{1}$$

For convenience, we denote the output probability vector by $f\left(x_{\leq i}\right)$:

$$x_{i+1} \sim f\left(x_{\leq i}\right) := \text{softmax}\left(f\left(x_{\leq i}\right)\right) := \text{softmax}\left(f\left(x_{\leq 0} \oplus x_1 \oplus \cdots \oplus x_i\right)\right), \tag{2}$$

where $a \oplus b = (a, b)$ is the concatenation of the vectors a and b and $x_{\leq i} := x_{\leq 0} \oplus x_1 \oplus \cdots \oplus x_i$.

For deterministic selection procedures, composing monotonic functions, such as softmax, is usually unnecessary. For example, the most likely next token is the $\arg\max$ of both the logits and the output of the softmax. Still, for convenience, we assume that LMs always output probability vectors. The sampling process in (2) is either deterministic (i.e., x_{i+1} is the token with maximal probability) or random (achieved by randomly selecting x_{i+1} from the distribution softmax $\left(f\left(x_{\leq i}\right)\right)$).

Speculative Inference (SI) is an approach for accelerating the inference of a *target* LM (e.g., a member of the GPT series) f_m. Such methods use faster LMs $f_1, \ldots, f_{m-1}$ that approximate the target model f_m in order to reduce the total inference time. For example, Leviathan et al. [2023] reduces the amount of time to infer a target model f_2 on a given prompt $x_{\leq 0}$ by using batching. SI algorithms start by drafting k tokens $x'_i := f_1(x'_{\leq i-1}) := f_1(x_{\leq 0} \oplus x'_1 \oplus \cdots \oplus x'_{i-1})$ $(i \in [k])$ using a faster drafter model f_1 before seeding the prompts $\{x'_{\leq i}\}_{i=0}^{k}$ altogether as one input batch to the target model f_2. The idea is to take advantage of the fact that modern GPUs can process the batch $\{x'_{\leq i}\}_{i=0}^{k}$ faster than feeding the k individual sequences independently.

Straightforward algorithms of speculative inference are typically *lossless in expectation*, i.e., they generate tokens from the same distributions as the target would generate without speculation. Naive algorithms of speculation guarantee returning the same tokens as the target [Joao Gante, 2023, Spector and Re, 2023]. More sophisticated algorithms of speculation might generate different tokens, but their generated tokens follow the distribution of the target [Leviathan et al., 2023, Chen et al., 2023].

To implement distributed algorithms for speculative inference, we use multiple **processors**, which are hardware components capable of **executing threads**. Processors can compute forward passes and sample tokens from the output probability vectors and we assume that threads can run in parallel. When using DSI we will run sequences of drafter models $f_{j_1}, f_{j_2}, \ldots, f_{j_k}$, where the first model takes $x_{\leq 0}$ and returns some token $x_1^{j_1}$, the second takes $x_{\leq 0} \oplus x_1^{j_1}$ as a prompt and returns $x_2^{j_1,j_2}$, and so on. As such, in order to denote that a given thread is computing the output of f_{j_k} on a sequence $x_{\leq k-1}^{j_1,\ldots,j_{k-1}} := x_{\leq 0} \oplus x_1^{j_1} \oplus \cdots \oplus x_{k-1}^{j_1,\ldots,j_{k-1}}$, we denote C_J, where $J = (j_1, \ldots, j_k)$. When a thread C_J computes an LM, we denote the output probability vector by C_J[prob]. If C_J samples a new token from C_J[prob], we denote this token by C_J[new]. For example, thread C_J implementing (2) above will have

$$C_J[\text{prompt}] := x_{\leq i},\ C_J[\text{prob}] := f\left(C_J[\text{prompt}]\right) \text{ and } C_J[\text{new}] \sim C_J[\text{prob}].$$

Once a thread C_J finishes sampling a new token, the thread outputs the concatenation of C_J[prompt] and C_J[new]. Following the example in (2), we have

$$C_J[\text{return}] := C_J[\text{prompt}] \oplus (C_J[\text{new}]) := (x_{\leq 0}, x_1, \ldots, x_{i+1}).$$

A new thread that was initiated by C_J is denoted by $C_{J\oplus(j)}$, where $J \oplus (j)$ is the concatenation of J and (j). The set of all the threads that originate from C_J is $\{C_{J\oplus J'} : J' \text{ is a nonempty tuple}\}$. We assume that terminating a concurrent thread terminates all the threads that originate from it.

Time in this paper is the *wall time*. We measure the time that passes from the initiation of a *task* until its termination. A task is a nonempty set of threads, denoted by $\{C_J : J \in \mathfrak{J}\}$. Its time is

$$T_{\text{wall}}\left[\{C_J\}_{J\in\mathfrak{J}}\right] := \max_{J\in\mathfrak{J}}(\text{Timepoint } C_J \text{ finishes}) - \min_{J\in\mathfrak{J}}(\text{Timepoint } C_J \text{ starts}).$$

When a task consists of a single thread, we omit the curly brackets, namely,

$$T_{\text{wall}}\left[C_J\right] := T_{\text{wall}}\left[\{C_J\}\right] \text{ where } |\{C_J\}| = 1.$$

Note that two threads, denoted by C_J and $C_{J'}$, may run concurrently and overlap in time. Hence, it is possible that $\max\{T_{\text{wall}}[C_J], T_{\text{wall}}[C_{J'}]\} \leq T_{\text{wall}}[\{C_J, C_{J'}\}] < T_{\text{wall}}[C_J] + T_{\text{wall}}[C_{J'}]$. However, if C_J and $C_{J'}$ do not overlap in time, then $T_{\text{wall}}[\{C_J, C_{J'}\}] \geq T_{\text{wall}}[C_J] + T_{\text{wall}}[C_{J'}]$.

3 Distributed Speculative Inference

While previous methods for SI [Leviathan et al., 2023, Chen et al., 2023, Miao et al., 2023] are useful for speeding up the inference, they overlook the idea of utilizing multiple processing units to compute LM outputs in parallel. In this section, we outline a theoretically sound approach to infer LMs using a sufficiently large number of processors. The naive version of our method operates under the assumption that we have access to a sufficient amount of processors so that threads never have to wait. Later, we discuss how our method can be implemented in practice with a fixed number of processors.

3.1 Method Overview

Consider the task of computing N output tokens autoregressively from a target model f_m given a prompt $x_{\leq 0}$. We have a set of faster drafter models, $f_1, \ldots, f_{m-1}$, that are all faster than f_m (as defined in Assumption 2). Our goal is to compute $x_i = f_m(x_{\leq i-1})$ for all $i \in [N]$. To achieve this, we initiate m threads, $C_{(1)}, \ldots, C_{(m)}$ (line 2). Each thread, denoted as (j_1), is responsible for computing $x_1^{j_1} = f_{j_1}(x_{\leq 0})$. Once a thread, $C_{(j_1)}$, finishes computation, we instantiate m new

Algorithm 1 Distributed Speculative Inference (DSI) of N tokens

```
Require: A prompt $x_{\leq 0}$, and $m$ autoregressive models, $f_1, f_2, \ldots, f_m$.
 1: $v = 1$.
 2: initiate $m$ threads $C_{(1)}, \ldots, C_{(m)}$ such that $C_{(j_1)}$ generates the token $x^{j_1} \sim f_{j_1}(x_{\leq 0})$ for all $j_1 \in [m]$ concurrently.
 3: label thread $C_{(m)}$ as the current verifier.
 4: ONCE any thread $C_{J\oplus(j)}$ finishes to generate a token, namely, sampled $C_{J\oplus(j)}[\text{new}] \sim f_j\left(C_{J\oplus(j)}[\text{prompt}]\right)$:
 5: if $|J| + 1 < N$ then
 6:     initiate $m$ threads, $C_{J\oplus(j,1)}, C_{J\oplus(j,2)}, \ldots, C_{J\oplus(j,m)}$, to generate a token concurrently and respectively from $f_1, f_2, \ldots, f_m$.
 7:     if $C_{J\oplus(j)}$ is the current verifier thread then
 8:         terminate all threads $C_{J\oplus(j')}$ (and their descendant threads) that sampled a different token than $C_{J\oplus(j)}$.
 9:         let $j^* = \arg\min_{j' \in [m]} \{j' \mid C_{J\oplus(j')}[\text{new}] = C_{J\oplus(j)}[\text{new}]\}$.
10:         terminate all threads $C_{J\oplus(j')}$ (and their descendant threads), where $j' > j^*$.
11:         label $C_{J\oplus(j^*,m)}$ as the current verifier.
12:         update $v = v + 1$.
13:         if $C_{J\oplus(j^*,m)}$ has already finished then
14:             go back to step 7 with $J = J \oplus (j^*, m)$.
15:         end if
16:     end if
17: else if the last entry of $J \oplus (j)$ equals $m$ (i.e., $j = m$) then
18:     return $C_{J\oplus(j)}[\text{return}]$.
19: end if
20: end ONCE
```

threads, $C_{(j_1,j_2)}$, to calculate $x_2^{j_1,j_2} = f_{j_2}(x_{\leq 0} \oplus x_1^{j_1})$ for all $j_2 \in [m]$. In general, once we compute $x_{r-1}^{j_1,\ldots,j_{r-1}}$, we initiate m new threads, $C_{(j_1,\ldots,j_{r-1},1)}, \ldots, C_{(j_1,\ldots,j_{r-1},m)}$, to compute $x_r^{j_1,\ldots,j_r} = f_{j_r}(x_{\leq 0} \oplus x_1^{j_1} \oplus \cdots \oplus x_{r-1}^{j_1,\ldots,j_{r-1}})$ for all $j_r \in [m]$. This is captured in lines 4 and 6.

Once $C_{(m)}$ completes its computation and provides the correct value of the first output token $x_1^m = x_1$, we can verify which other threads, $C_{(j_1)}$, have accurately computed x_1. Any thread $C_{(j_1)}$ where $x_1^{j_1} \neq x_1$ is immediately terminated along with its descendant processes. For each $j_1 \in [m]$ that correctly computed $x_1^{j_1} = x_1$, we continue with computing $x^{j_1,j_2} = f_{j_2}(x_{\leq 0} \oplus x_1^{j_1})$ for all $j_2 \in [m]$. However, since all threads are computing the same set of tokens, we terminate all but the one corresponding to the smallest value of j_1 that satisfies $x_1^{j_1} = x_1$. In essence, $C_{(m)}$ serves as a verifier, identifying drafters that miscalculated the initial part of the autoregressive computation. Once we retain one valid j_1, we relabel $C_{(j_1,m)}$ as the new verifier thread. We know that since $C_{(j_1)}$ returned the correct token $x_1^{j_1} = x_1$ and $x_2 = f_m(x_{\leq 1})$, the output of $C_{(j_1,m)}$ must be correct. When that thread finishes, among the remaining threads, $C_{(j_1,j_2)}$, we terminate those that miscalculated $x_2 = x_2^{j_1,m}$ and keep only the one with $x_2^{j_1,j_2} = x_2^{j_1,m} = x_2$, whose index j_2 is minimal. We continue this process until the output $x^{j_1,\cdots,j_{N-1},m}$ is obtained from the last verifier thread $C_{(j_1,\ldots,j_{N-1},m)}$. The process of relabeling verifier threads and terminating irrelevant threads is outlined in lines 8, 10, and 11. Line 13 considers the case where the newly labeled thread may have already finished. If so, in line 14, we return to line 7 with the new verifier thread.

Preemption in DSI. A crucial enhancement in DSI is the introduction of preemption. Algorithm 1 invokes a new process for each token. With preemption, DSI waits and processes a batch of tokens after a `lookahead` number of tokens have been drafted. This change reduces the number of invocations required, allowing the use of a fixed number of processing units. The `lookahead` parameter allows tuning DSI to use an arbitrary maximal number of available processing units. The maximal number of required targets is $\lceil \frac{\texttt{target\_latency}}{\texttt{lookahead}\cdot\texttt{drafter\_latency}} \rceil$. This approach maintains the integrity of the speculative inference algorithm, ensuring that the theoretical foundations hold while optimizing performance.

Rejection sampling algorithm. As can be seen in lines 8 and 10, Algorithm 1 rejects/terminates any thread (and its descendants) that returns a token that is not exactly the same as the token returned by the current verifier. However, this criterion is fairly strict and leads to many rejections in practical settings. Even if the drafter is another instance of the target, they may disagree due to the randomness of the sampling. In order to increase the number of acceptances while maintaining the distribution of the outputs of the target model, Leviathan et al. [2023], Miao et al. [2023] suggested different relaxed methods for rejecting draft outputs. In order to incorporate these rejection sampling methods, we can replace lines 8-9 with an application of their rejection sampling procedures.

3.2 Analysis

As a next step, we would like to prove that DSI (Algorithm 1) always returns the correct sequence of tokens $x_1, \ldots, x_N$ (it is lossless) and that it runs at least as fast as non-SI and SI. Before we state our main theoretical results, we state several assumptions that will be used in the analysis. The proofs are provided in Appendix A.

Assumption 1. *We assume the existence of a (universal) constant $c > 0$ such that, for any input prompt $x_{\leq 0}$ and model index $j \in [m]$, we have:*

$$T_{\textit{wall}}\left[\textit{computing } f_j\left(x_{\leq 0}\right)\right] \in (0, c) \quad \textit{and} \quad T_{\textit{wall}}\left[\textit{sampling } x \sim f_j\left(x_{\leq 0}\right)\right] = 0.$$

Assumption 2. *We assume that for all $j \in [m-1]$, f_j is faster than f_m (denoted $f_j \preceq f_m$) in the following sense $\max_{x_{\leq 0}} T_{\textit{wall}}\left[\textit{computing } f_1\left(x_{\leq 0}\right)\right] \leq \min_{x_{\leq 0}} T_{\textit{wall}}\left[\textit{computing } f_2\left(x_{\leq 0}\right)\right]$.*

Assumption 3. *We assume that $T_{\textit{wall}}\left[\{C_{(j_1,\ldots,j_i)}\}_{i=1}^{k}\right] = \sum_{i=1}^{k} T_{\textit{wall}}\left[C_{(j_1,\ldots,j_i)}\right]$.*

The first assumption asserts that computing the output of any model takes a non-zero, bounded amount of time, and sampling a token from the output probabilities takes a negligible amount of time. The second assumption asserts that each drafter model runs faster than the target model, for any given input prompt. The third assumption asserts that computing $x_k^{j_1,\ldots,j_k}$ takes the time of first computing $x_1^{j_1}$, then $x_2^{j_1,j_2}$, and so forth, up to $x_k^{j_1,\ldots,j_k}$, with no delays.

The following theorem suggests that our method returns tokens from the same distributions as those the target would generate without speculation, and is at least as fast as iteratively applying the target model itself.

Theorem 1. *Under Assumptions 1, 2 and 3, Algorithm 1 returns the same output and runs at least as fast as running the target model itself without speculative inference.*

Theorem 2. *Under Assumptions 1, 2 and 3, Algorithm 1 runs at least as fast as SI in expectation.*

The advantage of Algorithm 1 lies in its concurrency. The following example shows how DSI can accelerate the inference of a given target model using a drafter model that is faster than the target model and returns the correct output with high probability.

Proposition 1. *Suppose we have a drafter model f_1, a target model f_2 and a prompt $x_{\leq 0}$. Assume that f_1 requires t_1 time units to compute each of its outputs, and f_2 requires t_2 time units, where $t_2 > t_1$. Assume that given the prompt $x_{\leq i} = x_{\leq 0} \oplus x_1 \oplus \cdots \oplus x_i$, the probability that f_1 returns the (correct) token x_{i+1} is p. Then, the expected time it takes Algorithm 1 to calculate the correct*

Target	Drafter	Dataset	Target Latency (ms)	Drafter Latency (ms)	Drafter Latency (%)	Acceptance Rate (%)	Speedup DSI vs. SI
Vicuna-13B	Vicuna-68M	CNN-DM	37.7	2.5	6.5	63	1.47x
Vicuna-13B	Vicuna-68M	Alpaca	33.3	2.5	7.4	58	1.41x
Vicuna-7B	Vicuna-68M	CNN-DM	29.4	2.5	8.4	67	1.29x
Vicuna-7B	Vicuna-68M	Alpaca	26.0	2.5	9.5	59	1.70x
Starcoder-15B	Starcoder-168M	HumanEval	20.6	6.8	32.3	93	1.92x
Starcoder-15B	Starcoder-168M	MBPP	21.0	6.8	32.9	90	1.66x
Phi3-14B	Phi3-4B	HumanEval	52.1	34.0	65.3	95	1.41x
Phi3-14B	Phi3-4B	MBPP	52.2	34.3	65.8	94	1.37x
Phi3-14B	Phi3-4B	CNN-DM	52.4	34.6	66.0	93	1.39x
Phi3-14B	Phi3-4B	Alpaca	49.6	33.4	67.4	87	1.60x

Table 1: DSI Speedups over SI for various target/drafter pairs. We observe that DSI outperforms the SI implementation consistently across all models and tasks. These results are based on simulations with thread pools.

output is at most $t_1 p(N-1) + t_2((1-p)(N-1)+1)$ time units, compared to the $t_2 N$ time units required if we were to compute f_2 without speculative inference.

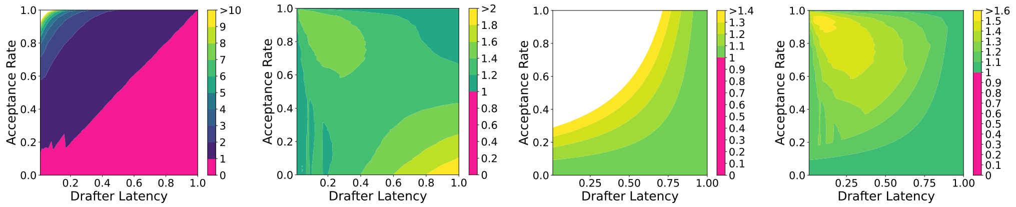
4 Experiments and Results

We conducted two sets of experiments to validate our theoretical results—that DSI outperforms SI [Leviathan et al., 2023] and non-SI (Theorems 1 and 2)—in practical settings.

Experiments with LLMs. The first experiment (see Table 1) studies the latency values and acceptance rates of pairs of off-the-shelf target and drafter LLMs on various well-established datasets. All the LLMs were downloaded from the Hugging Face Hub and used as-is. We evaluate our method on four datasets including three tasks: text summarization using CNN Daily Mail [Hermann et al., 2015]; instruction-following using Alpaca [Taori et al., 2023a]; and code generation using MBPP [Austin et al., 2021] and HumanEval [Chen et al., 2021]. For a complete description of the models, datasets and examples of relevant prompts, please refer to Appendix C and Appendix B.

For each combination of dataset d and corresponding target/drafter model f, we estimate the average latency of f in the following manner. First, we select 50 prompts from d uniformly at random, and for each prompt, generate 20 tokens using f, measuring the latency for each token in milliseconds. Following prior work, we distinguish between Time to First Token (TTFT) generation and Time Per Output Token (TPOT) generation (of all subsequent 19 tokens). Since TTFT is usually significantly longer than TPOT (which dominates the overall sequence generation time), all latency figures in Table 1 refer to TPOT, for brevity. Finally, we calculate the average TTFTs and TPOTs over all prompts per model/dataset pair, to estimate the expected latency of a single forward pass. Thus, the TPOT latency of the target LLM and the drafter LLM are shown in "Target Latency (ms)" and "Drafter Latency (ms)", respectively. We also report the ratio between the target and drafter latencies and present it in percentages ("Drafter Latency (%)").

In order to estimate the alignment level between each target and drafter pairs, we use the "Acceptance Rate" (AR). To calculate the AR, we generate 256 tokens using the drafter given the same prompt used by the target. For each prompt, we consider the lengths of the longest sequences of exact token matches between the target and the drafter. Below is a simplified example where tokens are counted as English words. If the target generates "We can only see a short distance ahead, but we can see plenty there that needs to be done. [...]" and the drafter generates "We can only see a short distance ahead, we done. [...]", then the longest sequence of exact matches is 8 tokens long. The expected number of accepted drafts is $\bar{n} := \frac{1}{N}\sum_i^N n_i$ where n_i is the number of accepted draft tokens for the ith prompt. The AR



(a) non-SI/SI **(b)** SI/DSI **(c)** non-SI/DSI **(d)** $\frac{\min(\text{SI,non-SI})}{\text{DSI}}$

Figure 1: Each heatmap (labeled "X/Y") plots the ratio between the run time of algorithm X and the run time of algorithm Y. See Appendix D for the detailed results. **(a)**: SI is slower than non-speculative inference (non-SI) when the drafter is either slow or inaccurate enough (pink marks slowdowns). DSI is never slower than either SI or non-SI. **(b, c, d)**: DSI is always faster than speculative inference (SI) and non-speculative inference (non-SI) algorithms for various drafters. **(d)**: DSI is up to 1.6x faster than the baseline algorithm, where the baseline is the faster between SI and non-SI. These results are based on simulations without thread pools.

is then calculated as $\texttt{acceptance\_rate} := 1 - \frac{1}{1+\bar{n}}$. We estimate latency and AR on a single A100 80GB GPU.

This experiment suggests that off-the-shelf LLM "families" such as StarCoder [Li et al., 2023] or Vicuna [Zheng et al., 2024] can form good pairs of target and drafter. Such families consists of LLM versions of different sizes that were trained in similar ways and on similar datasets. We notice that even relatively small drafters demonstrate good alignment with larger LLMs of the same family. For example, `Starcoder-168M` (drafter) and `Starcoder-15B` (target) yield an AR of 93%.

Experiments with thread pools. To measure the speedup of DSI relatively to SI (Speedup DSI vs. SI), we perform a simulation of generating 50 tokens using each target-drafter pairs on each dataset, using the latency and acceptance rate values computed above. The simulation of each combination above is run on multiple lookahead token values (namely 1, 5, and 10). For the DSI run, we perform a grid search over the lookahead tokens, but also different parallel target counts (namely 1 and 7). The DSI simulation involves opening a separate thread in parallel for each target, using python's thread-pool implementation. Overall, DSI outperforms SI consistently across all models and tasks.

Simulated SI without thread pools. Figure 1 presents the results of an experiment that simulates SI across a wide range of configurations. The aim of this experiment is to estimate pairwise speedups: DSI compared to SI, DSI compared to non-SI, and SI compared to non-SI. Since SI is slower than non-SI in some configurations, we have included an additional comparison that shows DSI speedups relative to the faster of the two algorithms—SI or non-SI—for any given configuration. This experiment helps identify configurations where DSI achieves the highest speedup. It demonstrates that, unlike SI, our method introduces no slowdown compared to non-SI and consistently accelerates inference, provided there are enough processing units. Furthermore, we demonstrate that our method remains useful in practical settings with a relatively small number of processing units.

In the experiment, the latency of DSI is computed using the formula from Lemma 1. This means that the `lookahead` parameter is set to one and that we need a maximum of $\texttt{ceil}(\frac{\texttt{target\_latency}}{\texttt{lookahead}\cdot\texttt{drafter\_latency}}) = \texttt{ceil}(\frac{\texttt{target\_latency}}{\texttt{drafter\_latency}})$ target servers (i.e., processors that are capable of computing the target). Having two target servers, for example, means that we may start computing the target on an input batch, even if the other instance of the target is not yet available. In practice, if we can compute the target LLM on a single GPU, then two target servers are two GPUs. If serving the target LLM takes two GPUs, then two target servers means we need four GPUs. For every configuration, we consider the latency of SI with $\texttt{lookahead} \in \{1, 2, \ldots, 200\}$

that minimizes its latency for that configuration. The configurations considered are the cartesian product of drafter latency $(0.01, 0.02, \ldots, 1)$, acceptance rate $(0, 0.01, 0.02, \ldots, 1)$, and `lookahead` $(1, 2, \ldots, 200)$. For each combination of (drafter latency, acceptance rate, and `lookahead`), we run 5 repeats of SI and average the results to estimate the expected latency of SI (the implementation of SI is described in Appendix D). Then, for each combination of (drafter latency, acceptance rate), we consider the minimal latency (letting SI select its optimal `lookahead`). Since the `lookahead` is a tunable parameter, our experiment assumes that it will be optimized by the user so that SI is optimized. It is known (and trivial) that SI is highly sensitive to the choice of the `lookahead` . To calculate the speedup of algorithm A over algorithm B per `(drafter_latency, acceptance_rate)`, we divide the latency of B by the latency of A. The speedups are not smooth for drafter latencies $< 20\%$ due to the discretization of the `lookahead` parameter. If we fix the `lookahead` and decrease the drafter latency to 0, the number of servers required by DSI grows to infinity. However, it is possible to tune the `lookahead` hyperparameter to an arbitrary number (as we did). For example, for `lookahead` $= 5$, the speedups are smooth for both algorithms (Figure 6).

As shown in Figure 1(a), to achieve a speedup with SI compared to non-SI, the acceptance rate of the drafter must at least match the latency of the drafter model (the region above the pink region in the figure). This means that the SI algorithm cannot speed up the inference if the acceptance rate of the drafter is not sufficiently high for a given latency. Conversely, in Figure 1(b), we observe that DSI consistently speeds up the inference time, regardless of the latency and acceptance rates of the drafter. This provides our method with much greater flexibility and robustness. In Figure 1(c), we observe that DSI is faster than non-SI for all the configurations for which non-SI is faster than SI. Finally, to obtain a comprehensive view of the inference speedup achieved by DSI, in Figure 1(d), we compare the performance of DSI with the minimal runtime of both SI and non-SI.

5 Discussion

In this work we studied how to reduce the run time of speculative inference algorithms by taking advantage of multiple processing units (e.g., GPUs). We have shown that in contrast to their empirical success, traditional SI algorithms can end up slowing the inference of LMs in various practical settings. For instance, when the drafters are insufficiently accurate or the drafter is too slow. We showed that by taking advantage of multiple GPUs, we can design a speculatively inference algorithm that provably reduces the inference time of both non-SI and SI algorithms. Our simulations validate our theory, indicating speedups for all possible configurations. For each configuration, the comparison is between DSI and the faster alternative algorithm (SI or non-SI). In essence this work paves the way to additional SI algorithms that can orchestrate multiple processing units at the same time.

Limitations. We introduce DSI and show that it is faster than SI and non-SI for all possible configurations by theoretical analysis and experiments. Our first experiment measures the time that it takes to compute LLMs. Then, the second experiment simulates DSI and SI. In the simulations, we replace the calls to LLMs with a wait. These wait times in the simulations are the expected wait times that we estimated in the first experiment. However, the simulations ignore latencies that exist in practice, such as the communication between processors (CPU and GPUs). Hence, the key limitation is that the algorithm is not yet implemented and tested over a physical computing node. Another limitation of DSI is the maximal number of servers that DSI requires. For example, if the target LLM fit a single GPU and the drafter latency is 14.29%, then DSI orchestrates a total of eight GPUs (seven instances of the target and one for the drafter). For faster drafters, DSI requires additional target servers. The exact number of servers is discussed in section 4. In our simulations with off-the-shelf LLMs (Table 1), we only consider configurations that do not require more than seven target servers.

Acknowledgements

We thank Intel Labs for funding this research.

References

Jacob Andreas. Language models as agent models. *arXiv preprint arXiv:2212.01681*, 2022.

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

Ankur Bapna, Naveen Arivazhagan, and Orhan Firat. Controlling computation versus quality for neural sequence models, 2020.

Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. A framework for the evaluation of code generation models. `https://github.com/bigcode-project/bigcode-evaluation-harness`, 2022.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

F. Warren Burton. Speculative computation, parallelism, and functional programming. *IEEE Transactions on Computers*, C-34(12):1190–1193, 1985. doi: 10.1109/TC.1985.6312218.

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.

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.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35: 30318–30332, 2022.

Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. Depth-adaptive transformer. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=SJg7KhVKPH`.

Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pages 10323–10337. PMLR, 2023.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In *Proc. of ICLR*, 2023. URL `https://arxiv.org/abs/2204.05999`.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. MiniLLM: Knowledge distillation of large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=5h0qf7IBZZ`.

Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis; Machine Intelligence*, 44(11):7436–7456, nov 2022. ISSN 1939-3539. doi: 10.1109/TPAMI.2021.3117837.

John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012. ISBN 978-0-12-383872-8.

Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. *Advances in neural information processing systems*, 28, 2015.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

Yen-Chang Hsu, Ting Hua, Sungen Chang, Qian Lou, Yilin Shen, and Hongxia Jin. Language model compression with weighted low-rank factorization. In *International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=uPv9Y3gmAI5`.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18(187):1–30, 2018.

Joao Gante. Assisted generation: a new direction toward low-latency text generation, 2023. URL `https://huggingface.co/blog/assisted-generation`.

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.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

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

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. In *MLSys*, 2024.

Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Ion Stoica, Zhijie Deng, Alvin Cheung, and Hao Zhang. Online speculative decoding. *arXiv preprint arXiv:2310.07177*, 2023.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. LLM-pruner: On the structural pruning of large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL `https://openreview.net/forum?id=J8Ajf9WfXP`.

Jonathan Mamou, Oren Pereg, Daniel Korat, Moshe Berchansky, Nadav Timor, Moshe Wasserblat, and Roy Schwartz. Accelerating speculative decoding using dynamic speculation length, 2024.

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:2305.09781v2*, 2023.

R OpenAI. Gpt-4 technical report. *arXiv*, pages 2303–08774, 2023.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Tal Schuster, Adam Fisch, Tommi Jaakkola, and Regina Barzilay. Consistent accelerated inference via confident adaptive transformers. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4962–4979, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.406. URL `https://aclanthology.org/2021.emnlp-main.406`.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Benjamin Spector and Chris Re. Accelerating llm inference with staged speculative decoding. *arXiv preprint arXiv:2308.04623*, 2023.

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

Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. In *The Twelfth International Conference on Learning Representations*, 2024a. URL `https://openreview.net/forum?id=PxoFut3dWW`.

Ziteng Sun, Ananda Theertha Suresh, Jae Hun Ro, Ahmad Beirami, Himanshu Jain, and Felix Yu. Spectr: Fast speculative decoding via optimal transport. *Advances in Neural Information Processing Systems*, 36, 2024b.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`, 2023a.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Alpaca: A strong, replicable instruction-following model. Stanford Center for Research on Foundation Models, 2023b. `https://crfm.stanford.edu/2023/03/13/alpaca.html`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Mingxue Xu, Yao Lei Xu, and Danilo P. Mandic. Tensorgpt: Efficient compression of the embedding layer in llms based on the tensor-train decomposition, 2023.

Zhewei Yao, Xiaoxia Wu, Cheng Li, Stephen Youn, and Yuxiong He. Exploring post-training quantization in llms from comprehensive study to low rank compensation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19377–19385, 2024.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36, 2024.

Yongchao Zhou, Kaifeng Lyu, Ankit Singh Rawat, Aditya Krishna Menon, Afshin Rostamizadeh, Sanjiv Kumar, Jean-François Kagy, and Rishabh Agarwal. Distillspec: Improving speculative decoding via knowledge distillation. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=rsY6J3ZaTF`.

A Proofs

Theorem 1. *Under Assumptions 1, 2 and 3, Algorithm 1 returns the same output and runs at least as fast as running the target model itself without speculative inference.*

Proof. We begin by demonstrating the losslessness of the algorithm. We would like to prove that when $v = k$, there is a thread C_{J_k}, that is the only thread that is labeled as a verifier, and it correctly computes the next token and that $J_k = J' \oplus (m)$ for some sequence $J' = (j_1, \dots, j_{k-1})$ of length $k-1$, where $x_i^{j_1,\dots,j_i} = x_i$ for all $i \in [k-1]$. We will prove this by induction on the value of v. In addition, we note that if this pattern is appreciated by the algorithm, then it is clearly a lossless algorithm.

Base case ($v = 1$)**:** Initially, when $v = 1$, there is only one verifier, $C_{(m)}$, which runs the target model f_m. Thus, when it finishes, it will return the correct token, x_1. Since the verifier is relabeled only when the value of v changes (see lines 11-12), as long as $v = 1$, the only thread labeled as a verifier is $C_{(m)}$.

Induction hypothesis: Assume that as long as $v = k$, there is only one thread C_{J_k} labeled as a verifier, which returns the correct token x_k, and that $J_k = J' \oplus (m)$ for some $J' = (j_1, \dots, j_{k-1})$ of length $k-1$, where $x_i^{j_1,\dots,j_i} = x_i$ for all $i \in [k-1]$.

Induction step: When v is updated from k to $k+1$, this change only occurs when the condition in line 7 is met. This condition indicates that the single verifier thread C_{J_k}, which is of length $|J_k| = k$, has finished computing its output token. By the induction hypothesis, this thread returns x_k as its output. Since f_m is slower than all drafter models $f_1, \dots, f_{m-1}$, all threads $C_{J' \oplus (i)}$ have already finished computing their outputs. Thus, when executing lines 8, 10, and 11, the only threads that remain active are the descendants of $C_{J' \oplus (j^*)}$, and the only thread serving as a verifier is $C_{J' \oplus (j^*, m)}$. Since $x_i^{j_1,\dots,j_i} = x_i$ for all $i \leq k-1$ and $x_k^{j_1,\dots,j_{k-1},j^*} = x_k$, then $C_{J' \oplus (j^*, m)}$ simply computes the output of the target model f_m on the correct sequence $x_{\leq 0} \oplus x_1 \oplus \dots \oplus x_k$. Hence, it correctly returns the $(k+1)$th token x_{k+1}, as desired.

Time: We notice that the algorithm terminates once it has computed the output of C_{J_N}. By Assumption 3, we have $T_{\text{wall}}\left[\text{Algorithm 1}\right] = \sum_{i=1}^{N} T_{\text{wall}}\left[\text{computing } f_{j_i}\left(x_{\leq i}\right)\right]$ and by Assumption 2, we have $T_{\text{wall}}\left[\text{computing } f_{j_i}\left(x_{\leq i}\right)\right] \leq T_{\text{wall}}\left[\text{computing } f_m\left(x_{\leq i}\right)\right]$. Together we obtain $T_{\text{wall}}\left[\text{Algorithm 1}\right] \leq \sum_{i=1}^{N} T_{\text{wall}}\left[\text{computing } f_m\left(x_{\leq i}\right)\right]$ which is the amount of time that it takes to compute the output tokens without speculative inference. □

Proposition 1. *Suppose we have a drafter model f_1, a target model f_2 and a prompt $x_{\leq 0}$. Assume that f_1 requires t_1 time units to compute each of its outputs, and f_2 requires t_2 time units, where $t_2 > t_1$. Assume that given the prompt $x_{\leq i} = x_{\leq 0} \oplus x_1 \oplus \dots \oplus x_i$, the probability that f_1 returns the (correct) token x_{i+1} is p. Then, the expected time it takes Algorithm 1 to calculate the correct output is at most $t_1 p(N-1) + t_2((1-p)(N-1)+1)$ time units, compared to the $t_2 N$ time units required if we were to compute f_2 without speculative inference.*

Proof. To understand how it works, let $j_1 \in \{1, 2\}$ be the smallest index such that $x_1^{j_1} = x_1$ and for all $i \in [N-1]$, we recursively define $j_i \in \{1, 2\}$ to be the smallest index such that $x_i^{j_1,\dots,j_i} = x_i$. We also fix $j_N = 2$. In addition, let $i_0 = 0$ and i_r be the rth index in $[N]$ such that $j_{i_r} = 2$. We notice that it takes $t_1(i_1 - 1) + t_2$ time units to compute the value of $x_{i_1}^{j_1,\dots,j_{i_1}}$. This is because we first compute x_1^1, then $x_1^{1,1}$, continuing up to $x_{i_1-1}^{1,\dots,1}$, and finally $x_{i_1}^{1,\dots,1,2}$. Each of the first $(i_1 - 1)$ tokens takes t_1 time units, while the final token takes t_2 time units. After $t_1(i_1 - 1) + t_2$ time units, we will have computed x_1^2, $x_2^{1,2}$, $x_3^{1,1,2}$, and so on, up to $x_{i_1}^{1,\dots,1,2}$. Since f_1 consistently generates accurate tokens up to index $i_1 - 1$, once we observe that x_1^2 matches x_1^1, we know that $x_2^{1,2} = x_2$ and can then verify that $x_2^{1,1} = x_2$ is also correct. Once we verify that $x_2^{1,1} = x_2$, we can verify $x_2^{1,1,2}$ and continue this pattern to verify $x_2^{1,1,1}$, and so forth. We note that calculating all of these tokens up to the calculation of $x_{i_1}^{1,\ldots,1,2}$ take at most $t_1(i_1 - 1) + t_2$ time units. Thus, we can verify that $x_{i_1}^{1,\ldots,1,2} = x_{i_1}$ with at most $t_1(i_1-1)+t_2$ time units. By the same argument as above, it takes $\sum_r (t_1((i_r - i_{r-1}) - 1) + t_2)$ time units to compute the value of $x_N^{j_1,\ldots,j_N}$ (and to verify its correctness). We notice that $Q = \sum_r (i_r - i_{r-1} - 1)$ is the number of indices $i \in [N-1]$ such that $j_i = 1$. Since $\mathbb{E}[Q] = p(N-1)$, we have $\mathbb{E}\left[\sum_r (t_1((i_r - i_{r-1}) - 1) + t_2)\right] = t_1 p(N-1) + t_2((1-p)(N-1)+1)$. □

Theorem 2. *Under Assumptions 1, 2 and 3, Algorithm 1 runs at least as fast as SI in expectation.*

Proof Idea. Consider the SI algorithm with `lookahead` $= k$ running on a drafter of latency c and a target of latency 1. At time 0, SI starts generating drafts, by its definition. At time kc, SI completes generating k draft tokens. SI completes verifying these drafts at time $kc+1$. If all of the k drafts are accepted, then we consider the $(k+1)$th returned by the target as accepted, by the definition of SI. Hence, SI starts generating the $(k+2)$th token at time $kc+1$ and is guaranteed to complete generating it at time $2 \cdot (kc+1)$, by the definition of SI. Since accepting tokens may accelerate the generation but cannot slow it down, the case of accepting all the first k tokens is the fastest. Hence, SI could not start drafting the $(k+2)$th token before time $kc+1$. It implies that SI could not complete generating the $(k+2)$th token before time $2 \cdot (kc+1)$. Consider a DSI algorithm that runs on the same pair of LLMs and uses `lookahead` $= k$ as well. Let $X_1, X_2, \ldots, X_k$ be indicator variables for the acceptances in the first speculative interaction of SI. That is, $X_i = 1 \iff$ the ith token is accepted. For DSI, define a new sequence $Y_1, Y_2, \ldots, Y_n$ such that $X_i = Y_i$ for all i. We saw that in case of $A_i = 1, \forall i$, SI generates the $(k+2)$th token after $2 \cdot (kc+1)$ time units. Unlike SI, DSI immediately starts drafting the $(k+1)$th token after completing generating the kth draft, by the definition of DSI. Namely, DSI does not wait on the verifier. Hence, in that case, DSI generates the first drafts at time kc, but it continues to generate the $(k+2)$th draft and completes verifying it at time $2 \cdot (kc) + 1$. If the $(k+1)$th draft got accepted, then the second batch that was sent to verification (at time $\leq 2 \cdot (kc)$) must include the sequence $x_{\leq 0} \oplus x_1 \oplus \ldots \oplus x_{k+1}$. Therefore, DSI generates the $(k+2)$th token at time $\leq 2 \cdot (kc) + 1$. For any nontrivial choice of c and k, DSI is faster. Otherwise, by the definition of DSI, it always has a non-SI privileged thread running. It is the sequence of current verifiers. Hence, at time 0, DSI runs the target on the initial prompt and receives back the first k tokens at times $\leq 1, 2, \ldots, k$ respectively. Let $n = \min\{i | A_i = 0\}$, the number of accepted drafts ($n \in \{0, 1, \ldots, k\}$). If $k \cdot c + 1 > n$, then the non-SI thread of DSI generates the first n tokens before SI. In that case, DSI starts processing the $(n+1)$th token before SI does. Otherwise, by the definition of DSI, both algorithms generate the first $n+1$ tokens at time $k \cdot c + 1$. The proof follows by induction. □

B Datasets and Prompts Details

We use standard datasets from Hugging Face and standard prompts from the state-of-the-art.

B.1 MBPP

MBPP dataset consists of crowd-sourced Python programming problems and is distributed under the cc-by-4.0 License.

Concerning the prompt, we followed [Ben Allal et al., 2022, Fried et al., 2023] and included the description of the programming task and a single test to verify solution, in order to help the model catch the signature of the function (see Figure 2).

```
"""{text}
{test_list[0]}
"""
```

Figure 2: MBPP Prompt

B.2 HumanEval

HumanEval dataset includes programming problems and is distributed under the MIT License.

Prompt contains only `prompt` field from the dataset.

B.3 CNN-DM

CNN-DM contains news articles and is distributed under the Apache License 2.0.

We included the `article` field in the prompt as in Figure 3.

```
"""Summarize:
{article}
Summary:
"""
```

Figure 3: CNN-DM Prompt

B.4 Alpaca

Alpaca dataset contains instructions and demonstrations. It is distributed under the cc-by-nc-4.0 License.

We follow Taori et al. [2023b] to define the prompts. For samples with a non-empty input field, we use the prompt as in Figure 4 while for samples with empty input field, we use the prompt as in Figure 5.

C Models

For all models, we retrieve model weights from Hugging Face. For clarity and reproducibility, we provide the URLs for each model used:

- `Vicuna-13B`: `https://huggingface.co/lmsys/vicuna-13b-v1.3`, distributed under Non-Commercial License.
- `Vicuna-7B`: `https://huggingface.co/lmsys/vicuna-7b-v1.3`, distributed under Non-Commercial License.
- `Vicuna-68M`: `https://huggingface.co/double7/vicuna-68m`, distributed under the Apache License 2.0.
- `Starcoder-15B`: `https://huggingface.co/bigcode/starcoder`, distributed under the Responsible AI License.
- `Starcoder-168M`: `https://huggingface.co/bigcode/tiny_starcoder_py`, also distributed under the Responsible AI License.
- `Phi3-14B`: `https://huggingface.co/microsoft/Phi-3-medium-128k-instruct` distributed under the MIT license.
- `Phi3-4B`: `https://huggingface.co/microsoft/Phi-3-mini-128k-instruct` distributed under the MIT license.

D Experiments Results

D.1 SI Implementation

```
def si(target_latency: float, drafter_latency: float, lookahead: int, N: int) -> float:
  total_cost: float = 0
  total_toks: int = 0
  while total_toks < N:
      num_accepted: int = get_num_accepted()
```

```
 """Below is an instruction that describes a
 task, paired with an input that provides
 further context. Write a response that
 appropriately completes the request.

### Instruction:
{instruction}

### Input:
{input}

### Response:
"""
```

Figure 4: Alpaca prompt for samples with a non-empty input field.

```
        total_toks += num_accepted + 1
        total_cost += lookahead * drafter_latency + target_latency
    return total_cost
```

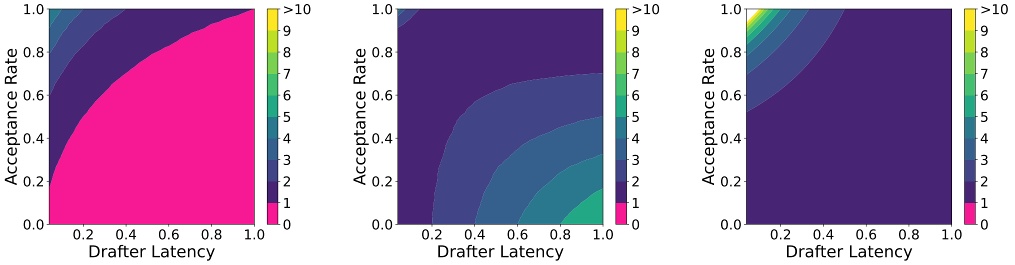
D.2 Speedups for `lookahead` = 5

```
"""Below is an instruction that describes a
task. Write a response that appropriately
completes the request.

### Instruction:
{instruction}

### Response:
"""
```

Figure 5: Alpaca prompt for samples with empty input field.



(a) SI/non-SI

(b) SI/DSI

(c) non-SI/DSI

Figure 6: Each heatmap (labeled "X/Y") plots the ratio between the run time of algorithm X and the run time of algorithm Y. SI is run with `lookahead` $= 5$. **(a)**: SI is slower than non-speculative inference (non-SI) when the drafter is either slow or inaccurate enough (pink marks slowdowns). DSI is never slower than either SI or non-SI. **(b, c)**: DSI is always faster than speculative inference (SI) and non-speculative inference (non-SI) algorithms for various drafters.