arXiv:2311.17065v3 [eess.AS] 10 Oct 2024

Turbocharge Speech Understanding with Pilot Inference

Rongxiang Wang
University of Virginia
waq9hw@virginia.edu

Felix Xiaozhu Lin
University of Virginia
felixlin@virginia.edu

ABSTRACT

Modern speech understanding (SU) runs a sophisticated pipeline: ingesting streaming voice input, the pipeline executes encoder-decoder based deep neural networks repeatedly; by doing so, the pipeline generates tentative outputs (called hypotheses), and periodically scores the hypotheses.

This paper sets to accelerate SU on resource-constrained edge devices. It takes a hybrid approach: to speed up on-device execution; to offload inputs that are beyond the device's capacity. While the approach is well-known, we address SU's unique challenges with novel techniques: (1) *late contextualization*, which executes a model's attentive encoder in parallel to the input ingestion; (2) *pilot inference*, which mitigates the SU pipeline's temporal load imbalance; (3) *autoregression offramps*, which evaluate offloading decisions based on pilot inferences and hypotheses.

Our techniques are compatible with existing speech models, pipelines, and frameworks; they can be applied independently or in combination. Our prototype, called PASU, is tested on Arm platforms with 6 – 8 cores: it delivers SOTA accuracy; it reduces the end-to-end latency by 2x and reduces the offloading needs by 2x.

CCS CONCEPTS

• **Information systems** → **Mobile information processing systems**; • **Computing methodologies** → **Machine learning**.

ACM Reference Format:
Rongxiang Wang and Felix Xiaozhu Lin. 2024. Turbocharge Speech Understanding with Pilot Inference. In *The 30th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '24), November 18–22, 2024, Washington D.C., DC, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3636534.3690694

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ACM MobiCom '24, November 18–22, 2024, Washington D.C., DC, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0489-5/24/11.
https://doi.org/10.1145/3636534.3690694

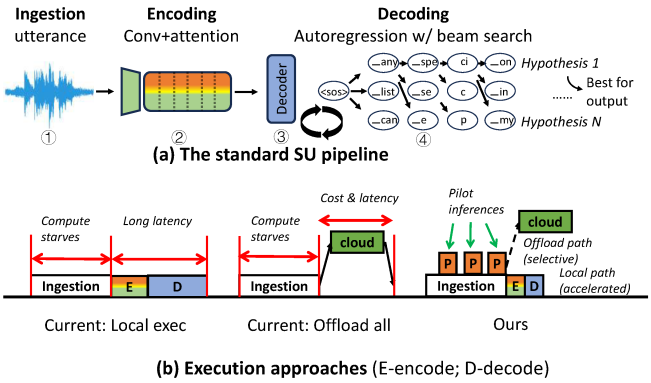

Figure 1: The SU pipeline and a comparison of execution approaches

Term	Definition
SU	Speech Understanding
Hypothesis	A tentative output, as a sequence of words or subwords
Autoregression	Generate an output sequence token by token
Beam search	A known method for generating parallel hypotheses in inference
CTC	A well-known method for scoring hypotheses in speech processing
Pilot inference (PI)	Our method of periodic executing inference on incomplete inputs

Table 1: A glossary of terms used in the paper

1 INTRODUCTION

Speech is a pervasive user interface for embedded devices. At its core are two speech understanding (SU) tasks: automatic speech recognition (ASR), transcribing voice to a sentence [47]; spoken language understanding (SLU), mapping voice to a structured intent, such as `{scenario: Calendar, action: Create_entry}` [6].

Modern SU: autoregression with beam search Modern SU runs a deep, encoder-decoder model [5] as shown in Figure 1(a). Ingesting an utterance waveform (①), the model comprises a neural encoder (②) and a neural decoder (③), generating a sequence of tokens, i.e. words or sub-words (④). This generation process is *autoregressive*: to produce a token, the decoder takes as the input all the latent units from the encoder, as well the output tokens it produces so far. To cope with noisy utterances, SU runs multiple decoding processes in parallel, each generating a candidate output sequence (called a hypothesis). SU picks the best sequence as the final output. Thanks to the encoder/decoder wrapped in beam search, modern SU can understand natural utterances

(e.g. "I need to turn on light in bedroom ") [3] beyond simple ones (e.g. "light on"). This paper focuses on encoder-decoder models, referring to them as deep SU.

Deep SU models are resource-hungry, often requiring GBs of memory and tens of GFLOPs per input second [1]. As a result, a model can take a few seconds to process a typical voice command. Facing the constraints, many embedded devices chose to offload all voices to the cloud [28], which, however, raises concerns on high cloud cost [17, 27, 33], privacy [36], latency [24], and service availability.

Problem & approach This paper presents an engine to accelerate deep SU on resource-constrained embedded devices[1]. The engine comprises a local execution path and an offloading path. While the local/offloading approach is reminiscent of many ML systems [22, 37, 46], our design specifically addresses unique challenges from speech.

1.1 The local execution path

Challenge: temporal load imbalance We identify a key inefficiency in the Speech Understanding (SU) task: poor resource utilization during the ingestion phase. The state-of-the-art attention-based SU system necessitates the compute to remain idle during the ingestion period, creating a surge of encoding-decoding workload after receiving the complete data, as illustrated in Figure 1(b). Incomplete data can significantly compromise accuracy for these systems. Similar accuracy degradation is also observed in streaming SU systems, as discussed in Section 8.

In summary, the device needs to utilize ingestion-time compute resources, in order to accelerate the expensive processing that can only start after the ingestion.

Our idea: pilot inference During ingestion, SU periodically encodes and decodes the incomplete input the device has received so far. This is shown in Figure 1 (b). Our key insight is that the the pilot inference's *intermediate state* can assist the full inference (i.e. the inference executed after ingestion, over the complete input). With the idea, we present multiple techniques: (1) beam collapse: the full inference only needs to verify its beam search path against the path in the pilot inference, and only falls back to full beam search in case of path divergence; (2) early termination: extrapolating the output length of pilot inference, the full inference predicts the final output length and help with beam search early termination; (3) CTC leap: the full inference approximates the connectionist temporal classification (CTC) prefix scoring [41] with the scores pre-computed by pilot inferences.

Pilot inferences are incremental in nature: the $(i+1)$-th inference instance reuses of the state from the i-th instance, in the same way as the full inference reuses the last pilot inference instance. In this way, the total cost of pilot inference is amortized over the duration of ingestion, and scales gracefully with the input length.

Pilot inference (PI) is related to recent speculative execution for language models [15] with a key difference: while the later optimizes inference on *complete* inputs, PI processes successive *incomplete* inputs, for which it addresses different challenges. Section 8 presents a detailed comparison.

1.2 The offloading path

Challenge: Lacking offramps for autoregression Typically, to decide whether to offload an input x, the device assesses its *confidence* on x, i.e. how likely the on-device inference yields a sufficiently accurate answer. Such an assessment is referred to as *offramps* in prior work. Existing offramp designs typically evaluates confidence as an ML model's prediction entropy [43, 49]. For SU however, existing offramps mismatch in two ways.

First is the *confidence measure*. Unlike a classification task which evaluate a model's layers sequentially and provides a single output that represent the probability, SU's output generation is both concurrent and iterative: it produces multiple hypotheses of probabilistic tokens sequence in a token-by-token fashion; in each iteration the decoder will generate the next tokens' probabilities base on current hypotheses. It was unclear how to derive a single confidence score out of an ongoing generation process.

Second is *timing*. Given an input, offloading represents a one-shot decision that can be taken at various times throughout the SU pipeline shown in Figure 1. While a deferred decision would have more input information for measuring the model confidence, it also incurs on-device delays. The timing therefore hinges on tradeoffs between an offramp's selectivity and overhead.

Our idea: PI-aid perplexity-based offloading The device evaluates its offloading decision as soon as it finishes ingestion, based on the perplexity score [13] of the last pilot inference. Doing so strikes a sweet-spot: (1) as the discrepancy between the last pilot inference and the full inference is small, their perplexity scores are also similar; (2) the device does not need to hold off the offloading decision until the completion of full inference. Notably, offloading an *incomplete* input (i.e. before the ingestion completes) does not speed up offloading the complete input: as an input utterance is no more than tens of KB, offloading is typically bound by a network round trip. With our PI-aid perplexity-based offloading, the system can achieve the target accuracy with reduced latency and offloading cost.

[1] shortened as "devices", to be defined in Section 2

1.3 Results and contributions

We implement a system called PASU (Pilot-Aid SU), atop two embedded platforms with 6 and 8 Arm64 cores respectively. We test PASU on SLURP [3], a challenging speech benchmark comprising 102 hours of speech. On both ASR and SLU tasks, PASU delivers strong accuracies with end-to-end latencies as low as 100s of ms, which is close to or below the threshold for human latency perception. Compared to popular, efficiency-optimized on-device models [10, 30], PASU's on-device execution incurs 2x lower latency with little energy overhead.

Combining local execution with offloading, PASU's hybrid execution offers much higher accuracy than on-device models at similar or lower latency; the hybrid execution offers close-to-gold accuracy (only 1% below SOTA), while reducing the offloading needs by 47% – 53%.

Towards deep SU on embedded devices, we contribute:

- PASU, a first-of-its-kind hybrid SU engine specifically optimizes for embedded devices.
- Pilot inference, a novel design to exploit under-utilized resources during ingestion.
- A new offramp design that assesses model confidence in generative, autoregressive output. This is the first mechanism for offloading generative tasks, to the best of our knowledge.

We will make our model and code publicly available.

2 MOTIVATIONS

2.1 Speech understanding (SU) at the edge

An unsolved mission On simple utterances, classic ML already attained word error rate (WER) as low as 4% on LibriSpeech [29]. On real-life speech as in the SLURP dataset [3], however, WER can be as high as 20% [11]. Deep models reduce WER significantly, albeit requiring much more compute resources.

Our system model A device has several CPU cores (no more than 8) and around 100 MB of memory budget [18, 19]. The device is wirelessly connected to the cloud speech services; yet, fewer cloud invocations are preferred, as they incur monetary cost and increase privacy risks. Users expect low latency (below a few hundred milliseconds) after they finish speaking to the devices. The device detects the beginning and end of an utterance with well-known, lightweight algorithms [21, 25].

2.2 A primer on SU pipelines

As shown in Figure 2, a pipeline consists of two modules.

Module 1: The encoding process Typically, the encoder in an SU model consists of multiple layers that combine convolution and attention-based elements. This design can be found in models such as Conformer [10], Branchformer [30], Wav2vec [2], and HuBERT [12]. During operation, as shown in Figure 2 left part, the 1-D speech data will be broke in to frames and undergoes STFT and Mel-filter bank methods to create a 2D spectrum that have $f_1 ... f_{T_o}$, each f_i is the processed frequency component of that time frame. The 2D spectrum $[f_1, ..., f_{T_o}]$ is then processed by a few convolutional layers for downsampling, $X = x_1, ..., x_T$ is get from $X = subsampling([f_1, ..., f_{T_o}])$. The X is subsequently passes through multiple encoder layers that include both convolutional and transformer component and get the latent speech representation Z among the same time frames $z_1, ..., z_T$ like X, $Z = encoders(X)$. This entire encoding process is a one-pass operation, providing an intermediate representation of the speech signal framewise with a specific dimension.

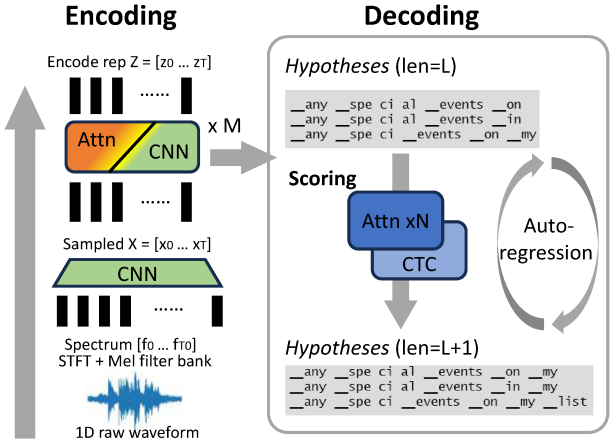

Figure 2: The deep SU pipeline [2, 10, 30]. The input of SU tasks, an 1D raw wavform of the audio, will first go through STFT+Mel filter bank to get spectrum, then processed by encoder to get intermediate encode representation, and finally decoded in an autoregressive fashion by hybrid transformer/CTC decoder coupled with beam search method

Module 2: The decoding process The generation of the final transcript involves an autoregressive beam search process as shown in Figure 2 right part, with tokens generated one by one. The beam search employs a specific beam width (k), maintaining k developing hypotheses during the decoding process. In decoding, the CTC decoder first calculates posterior token probabilities $p(z_i = c|Z)$ for each frame z_i of the encoder output. Beginning with the "<sos>" (start of the sentence) token, the transformer decoder calculates the next token c's posterior probabilities based on current partial hypothesis g and speech latent representation and get $p_{\text{attn}}(c|g, Z)$, combine with the previous token probability we have $p_{\text{attn}}((g, c)|Z)$.

Throughout decoding, hypotheses are scored by a combination of the transformer and CTC prefix scorers [41]. Prior work deem both scorers vital: the transformer exploits language features; CTC exploits acoustic features.

Notably, the CTC scorer can be slow, as its dynamic programming algorithm shows lower parallelism, barely benefiting from modern CPU/GPU. It calculates CTC prefix probabilities $p_{\text{ctc}}((g, c, ...)|Z)$ based on newly developed hypotheses (g, c) and the CTC framewise output $p(z_i = c|Z)$. The best K hypotheses with highest $\lambda \log p_{\text{attn}}((g, c)|Z) + (1 - \lambda) \log p_{\text{ctc}}((g, c, ...)|Z)$ are retained in the beam for the subsequent rounds of decoding, based on a weighted sum of probabilities from the transformer decoder and CTC prefix scorer. The decoding process concludes when the completed hypothesis significantly outperforms the developing hypotheses, as described in [41].

We next analyze the inefficiencies in the two modules.

2.3 The encoding inefficiency

First, the ingestion is streaming. A typical spoken input lasts as long as 3–5 seconds [3, 29]. In contrast to vision and NLP systems which usually ingest complete images or sentences, an SU system receives an input utterance by pieces.

Second, the input completeness matters to accuracy. The information in the input audio is distributed throughout its entire duration, with the entirety of the audio serving as the context for each individual part. While waiting for data ingestion may seem to hinder resource efficiency, it is a necessity for attention models. The all-to-all attention mechanism in transformer-based models is the key to their superiority in NLP and speech-related tasks. This mechanism leverages contextual information from the complete dataset to generate results with better language consistency. As shown in prior work [5, 10, 30], any portion of the output should take into account the whole input (i.e. the context). As an anecdote, in time-related phrases, words like "calendar" or "reminder" should be assigned with higher probabilities. Partial input data with incomplete context can significantly impair the model's decoding accuracy. As such, a common practice is for the models to wait for ingestion completion [32].

As a result, throughout a deep SU pipeline the resource utilization is severely imbalanced. The compute remains idle during ingestion, while becoming busy during the time-consuming encoding and decoding processes. This motivates us to overcome the aforementioned algorithmic constraint, shifting a fraction of computation to the ingestion. We expect only marginal increase in energy consumption, as the device is already busy processing audio IO. We will validate energy in Section 7.

2.4 The decoding inefficiency

Algorithmically, SU decoding exhibits high complexity. If we use NFE to denote the number of neural function evaluations in generating one token, the total decoding complexity is roughly $O(k \times \ell \times NFE)$, where the beam width k is the number of parallel hypotheses, and the depth ℓ is the max hypothesis length before the search termination. As an example, an input of 3 seconds would require 50—125 NFEs.

Much computation is likely redundant. Notably, the decoding process generates hypotheses much longer than the best hypothesis it eventually selected as the output: on popular benchmarks, we observe the average ℓ is ~18, while the true output length is ~11 on average, suggesting up to 40% NFEs could have been saved.

Empirically, the decoding is slow on typical edge devices. As section 7 will show, a modern Arm SoC takes around 1 second to process a typical voice command, far exceeding user's perception threshold at a few hundred milliseconds. Unfortunately, mobile GPUs would not help much. Our experiments on Nvidia's Ampere mobile GPUs are even 1.6x slower than CPUs on the same board (see Table 2, 6CB). This is likely due to the irregular decoding workloads, especially the CTC scorer, leaving the GPU hardware parallelism underutilized. Long decoding latency also keeps the device SoC in a high power mode, consuming more energy.

3 PASU OVERVIEW

3.1 The on-device SU model

As prerequisites to system design, we refactor a typical SU model. This is shown in Figure 3 (a).

Encoder: late contextualization We ensure that most of encoding layers can execute in parallel to ingestion. Rather than computing all-to-all attention at each encoding layer, we make the bottom layers (closer to the input) compute only convolution, and the few top layers compute attention.

Decoder PASU has separate decoder instances for pilot inferences and for the full inference. The two instances share the same structure and weights, whereas the pilot decoder runs a narrower search beam.

The model details can be found in Section 6.

3.2 The system

Ingestion As shown in Figure 3(b)., PASU ingests segments of voice input. Upon the arrival of each segment, PASU executes the convolutional encoder layers, producing per-segment results (①). Periodically, PASU executes pilot inference (②): combining the per-segment results accumulated so far and sending them to the attentive encoding layers, followed by the decoding process that generates a sequence of tentative tokens, which we refer to as a "pilot output".

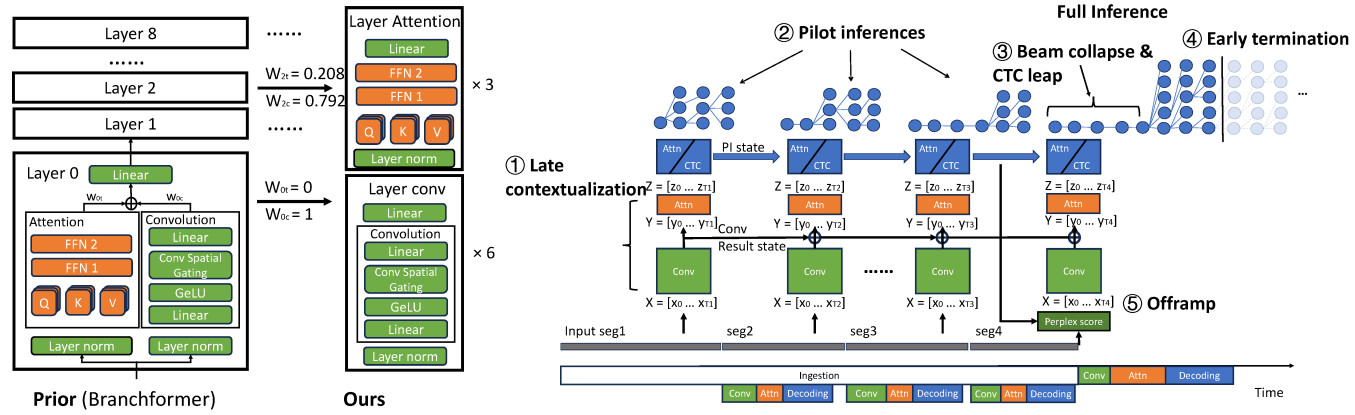

Figure 3: Overview. (a) Compared to a popular model (left), our model (right) shifts much of the encoding computation to bottom layers, allowing them to be executed in a streaming fashion during ingestion. (b) Execution pipeline of PASU. Our system performs pilot inference periodically on partial input, and use the pilot inference result to speedup the final inference

Pilot inference Over the course of ingesting an input, PASU repeats pilot inference multiple times, each time on an increasingly longer input. The frequency of pilot inferences is a configuration parameter to be evaluated experimentally. Essentially, pilot inferences extract valuable information from the partial inputs, *during* the ingestion. This information speeds up full inference, which benefits the local execution path; it also contributes to confidence estimation, which benefits the offloading path.

Offramp Once the ingestion is over (e.g. a detected pause [21]), PASU evaluates the offloading decision immediately, by computing the confidence score of the last pilot output (⑤). By doing so, PASU makes the decision without waiting for inference on the complete input $\{x_{i=1..T}\}$; since the last pilot inference is based on the longest partial input, the resultant confidence is expected to approximate to that in decoding the full input.

If the confidence is low, PASU offloads the input as waveform and waits for results; otherwise, it encodes the complete input and does full decoding locally. Note that: offloading intermediate results (as opposed to waveforms) often saves no network traffic, as each waveform is often tens of KBs. PASU refrains from emitting the pilot output to the user, because of high word errors. Nevertheless, the pilot output benefits decoding in crucial ways: to reduce the beam search width (③), to approximate the CTC prefix scores (③), and to help with the beam search early termination (④).

3.3 Applicability

Our techniques can be applied independently or in combination to existing SU. (1) Late contextualization and pilot inference both speed up local execution; they can be applied separately to existing models. Specifically, pilot inference can be applied to *unmodified* models, such as Branchformer and HuBERT. Notably, it is compatible with both attention decoding (such as in Whisper [34]) and hybrid CTC/attention decoding, as we will show in Section 7. (2) Our offramps, which introduce offloading to SU, can be applied to other on-device SU models in order to allow local/cloud hybrid execution. Without our design in (1) however, offloading decision would have to be evaluated on full decoding sequences, incurring extra delays.

Our techniques require light modifications to the existing SU implementations. It does not requires to develop new ML operators. Late contextualization requires to train a new model architecture; the model can run atop existing ML libraries and toolkits such as ONNX runtime and ESPnet. Pilot inference and offramps modify the SU pipeline logic, which is implemented in Python code on CPU.

Our system generally targets commodity mobile devices like smartphones and laptops. We expect these devices to be capable enough to execute the on-device model in PASU faster than offloading to the cloud. For more constrained devices, a customized, lighter-weight on-device model can still help orchestrate the system, though this may require compromising on local processing accuracy.

4 THE LOCAL EXECUTION PATH

This section focuses on pilot inference for local execution.

4.1 The mechanisms

Pilot inference works periodically on partial data as shown in Figure 3(b) (②). PASU refrains from performing pilot inference on excessively short data as it is hard to generate meaningful result with too short partial data. During data ingestion starting from T_0, the system conducts pilot inference on partial data periodically with a pre-defined granularity of δt, intended to complete within δt. In each round of pilot inference, the system utilizes the same encoder and the pilot decoder to engage in beam search inference with a beam size of k. Supposing the pilot inference is performed on data length $T_0 + n\delta t$, it will take at most δt to finish and the results cover the full data that have length range between $[T_0 + (n+1)\delta t,\ T_0 + (n+2)\delta t]$. To manage the runtime of pilot inference, a token limit of ℓ tokens is set. Following the decoding of ℓ tokens, the hypothesis with the highest probability is chosen as the reference hypothesis for local execution or hybrid execution path decisions. The reference hypothesis, denoted as $yp_{1:n} = (yp_1, \ldots, yp_n)$, offers the potential to accelerate the full inference process from multiple perspectives mentioned in the next few subsections.

Hyperparamters Several key parameters in pilot inference require pre-operation definition. These parameters include granularity δt, beam size k, and the maximum token length for beam search inference, denoted as ℓ. These hyperparameters are determined through heuristic methods based on experimental experience. The granularity needs to be established initially. Typically, the system anticipates that the longest partial data covers at least 70% of the full-length data. To achieve this, the granularity is expected to satisfy $\delta t < 0.2T$. It's important to note that the granularity doesn't have to remain a fixed number throughout system operation. As data lengthens, the pilot inference take longer time as shown in Figure 3(b). (②) comparing the different pilot inference, the granularity can also expand. The beam size for pilot inference is set at 60% of the beam size used for full decoding to ensure lightweight pilot inference while maintaining fidelity. The inference length is determined based on average inference length. If the average inference length is ℓ_{full}, PASU sets ℓ to be $0.7 \cdot \ell_{\text{full}}$.

Incremental execution The pilot inference based full decoding acceleration techniques (which will be elaborated in the following sub-sections) can also be adopted in the pilot inference execution. More specifically, the reference based beam collapse can be adopted in the pilot inference. In our system, as shown in Figure 3(b). (②), from the second pilot inference, each pilot inference can take reference hypothesis from the previous pilot inference and perform all the optimization in the pilot beam search procedure. Incremental execution is critical for pilot inference runtime reduction.

4.2 Optimization 1: beam collapse

Assumption Most tokens of the full transcript should be the best token in the beam search decoding process at the corresponding round. For the beginning part of the transcript, the reference transcript from pilot inference should share most of the tokens the same with the final transcript. **What we do** Base on this assumption, during full inference, starting from the first token, in each inference round before predicting the next token probability using the scorers, the system attempts to validate the reference hypothesis. It checks the latest token in the current best hypothesis, y_i. If $y_i = yp_i$, the token is validated, and the system retains only the best hypothesis for the subsequent token calculation in this round. **Complexity reduction** This verification reduces the number of running hypothesis for predicting the next token to 1, also reduces required NFE to 1. **Why such approximation is acceptable** The approximation is reasonable because of two point: the reference result is from pilot inference works on most part of the data by design. The verification step also partially secures the inference not to be deviated by the potential wrong part in the reference transcript.

4.3 Optimization 2: early termination of beam search

Assumption The reference inference results can reflect the final transcript length, as the token number grows proportionally as time grows. **What we do** The system predicts the final inference length using three key parameters: the length of the partial data, ℓ_{partial} obtained in the latest pilot inference; the full data length, ℓ_{full}; and the token count of the reference hypothesis, n_p. The calculation for the full inference length is expressed as $n = \ell_{\text{full}}/\ell_{\text{partial}} \cdot n_p + C$. C represents a tunable constant that loose the early termination in full inference. During the full inference, when the full inference reaches the predicted length n, if there are terminated hypotheses, the system terminates the beam search inference. Otherwise, the system continues inference until at least one hypothesis ends before terminating the process. **Complexity reduction** Assume the vanilla beam search length is ℓ_v, the early termination can save $(\ell_v - n)$ round of NFE calculation. **Why such approximation is acceptable** The approximation works well because the tokens distribute evenly among time statistically. The loose factor C can also help to decrease the deletion error that introduced by too aggressive early termination.

4.4 Optimization 3: fast prefix scoring with CTC leap

Assumption Some of the CTC prefix scoring computation during pilot inference can be reused in the full inference process. The original CTC prefix scoring requires recursively

Algorithm 1: CTC prefix scoring with CTC leap

1: **function** α_{ctc} h, X , $rp^{(n)}_{[1...T_{\text{pilot}}]}(h), rp^{(b)}_{[1...T_{\text{pilot}}]}(h)$
2: g,c ← h # split h into the last label c and the rest g
3: **if** c = <eos> **then**
4: **return** $\log\{r_T^{(n)}(g) + r_T^{(b)}(g)\}$
5: **else**
6: $r_1^{(n)}(h) \leftarrow \begin{cases} p(z_1 = c|X) & if\ g =< sos > \\ 0 & otherwise \end{cases}$
7: $r_1^{(b)}(h) \leftarrow 0$
8: $\Psi \leftarrow r_1^{(n)}(h)$
9: $\Phi_{[1...T-1]} \leftarrow r^{(b)}_{[1...T-1]}(g) + r^{(b)}_{[1...T-1]}(g) \cdot (last(g) == c)$
10: $r^{(n,b)}_{[1...\text{ int}(T_{\text{pilot}}\cdot q)]}(h) \leftarrow rp^{(n,b)}_{[1...\text{ int}(T_{\text{pilot}}\cdot q)]}(h)$
11: **for** t = int($T_{\text{pilot}} \cdot q + 1$) ... T **do**
12: $r_t^{(n)}(h) \leftarrow \left(r_{t-1}^{(n)}(h) + \Phi_{t-1}\right) p(z_t = c|X)$
13: $r_t^{(b)}(h) \leftarrow \left(r_{t-1}^{(b)}(h) + r_{t-1}^{(n)}(h)\right) p(z_t =< b > |X)$
14: **end for**
15: $\Psi \leftarrow \Psi + \text{sum}(\Phi_{[1...T-1]} \cdot [p(z_2 = c|X) \dots p(z_T = c|X)])$
16: **return** $\log(\Psi)$
17: **end if**
18: **end function**

Algorithm 1: Algorithm modified with CTC leap for CTC prefix scoring speedup. The highlighted part is our modification to reuse the calculation in pilot inference and skip part of the calculation. The remaining parts are the same as the hybrid CTC/attention algorithm [41].

calculated among all the time frame $[1, T]$. In pilot inference, calculation among $[1, T_{\text{pilot}}]$ has been done. The system just need to finish the calculation among $[T_{\text{pilot}}, T]$ **What we do** We propose CTC leap to help accelerate the CTC prefix scoring step in Hybrid CTC/Attention decoding (HD) algorithm [41] by reusing the computation in pilot inference. The algorithm modified with CTC leap is shown in algorithm 1 The CTC prefix scoring in HD algorithm is a modified version of forward algorithm that help calculate the prefix CTC probabilities among all the time frames. In the algorithm, the prefix forward probability of prefix h among 1 to t time frames is denoted as $r_t^{(n)}(h)$ and $r_t^{(b)}(h)$, for the prefix end with non-blank and blank token. In the algorithm, the prefix h will be divided into the last token c, which is the new token that to be developed, and the prefix part g. basically (g, c) equals h. If c is the <eos> token, same with original algorithm, $r_T^{(n)}(g) + r_T^{(b)}(g)$ by definition is the final overall probability result . When c is other tokens, the $r^{(n)}_{[1...T]}(h)$ and $r^{(b)}_{[1...T]}(h)$ is calculated through the loop between line 11-14 recursively. The ψ is the final cumulative prefix probability among all the time frame, calculated based on the $r^{(n)}_{[1...T]}(g)$ and $r^{(b)}_{[1...T]}(g)$ and the CTC posterior probability $p(z_t = c|X)$. In the calculation, the bottleneck is the recursive loop calculation of $r^{(n)}_{[1...T]}(h)$ and $r^{(b)}_{[1...T]}(h)$. Different from the original algorithm, as high lighted in algorithm 1, in our algorithm, we reuse the $r^{(n)}_{[1...T_{\text{pilot}}\cdot q]}(h)$ and $r^{(b)}_{[1...T_{\text{pilot}}\cdot q]}(h)$ from pilot inference to skip the calculation from time frame 1 to $T_{\text{pilot}} \cdot q$ and only do $T_{\text{pilot}} \cdot q$ to T when beam collapse happens. $q \in [0.5, 1]$ is a factor that help discard the end part of the time frames. This could work because if the beam collapse happens, the reference hypothesis from pilot inference should also include prefix g, as well as the calculation result of $r^{(n)}_{[1...T_{\text{pilot}}\cdot q]}(h)$ and $r^{(b)}_{[1...T_{\text{pilot}}\cdot q]}(h)$. **Complexity reduction** When the speedup requirement is met (pilot inference token is verified), the CTC prefix probability calculation in this round can be saved by $T_{\text{pilot}} \cdot q/T \cdot NFE_{\text{CTC}}$. **Why such approximation is acceptable** We reused the $r^{(n)}_{[1...Tpilot\cdot q]}(h)$ and $r^{(b)}_{[1...T_{\text{pilot}}\cdot q]}(h)$ in the pilot inference. We argue that the reused computation result is a good approximation. It is because essentially the reused calculation are (1) ultimately calculated from CTC posterior $p(z_t = c|X)$, which is calculated framewisely with CTC linear layer, thus the beginning part does not depend on the latter part. (2) the reused calculation is also based on a forward recursive algorithm, make it independent to the latter time frame.

4.5 Implications of approximation

Approximate execution is common to SU and more general autoregressive system like LLM, because the whole thing is statistically in nature; and people often make approximate at their discretion, as long as the approximation can be empirically validated. The speculative decoding technique in [15] also involves tentative hypothesis validation. The beam search end estimation technique in [14] introduce tunable parameter "patience factor" to help tune the beam search length. The truncate CTC prefix scoring in [26] estimate the CTC prefix alignment with time frame to save part of the computation. Different from these method, our system can better work on these estimation with the help of reference result from pilot inference.

5 THE OFFLOADING PATH

While the above designs accelerate the local execution, the SU task accuracy is nevertheless bound by the model size on the device. To deliver SOTA accuracy, we further augment the SU pipeline with cloud execution. The key challenges are twofold. (1) The device should estimate confidence (as the offloading criteria). Existing method [43] used in classification tasks cannot be directly applied to the generative SU task. It is because it works with the BERT style encoder, only take first frame of the output for classification and cannot take the full input into consideration. (2) The device should do so without delaying the local processing. It cannot, for example, simply evaluate the decision *after* full inference, as this would result in extended local decoding times for all

inputs. It also cannot offload *prior to* ingestion completion: as each voice input is typically tens of KB to ~300 KB for speech less than 10 s, the offloading is bound by network round trips (RTTs); the device would still need one RTT to offload the whole input, after the ingestion completes.

Confidence estimation *Perplexing Score-Based Approach* The perplexing score is derived from the reciprocal of the average probability of all tokens in the hypothesis, calculated by $exp(-\frac{1}{\ell}\sum_{i=0}^{\ell}\log(p(y_i|y_{0..i-1})))$. A high perplexing score indicates low confidence of the local model with the data. The system sets different thresholds for the perplexing score and offloads data that surpasses these thresholds.

RNN-Based Approach During beam search, the system gathers the probability of each token in the hypothesis. This fine-grained probability information better represents the local confidence in the data. Different patterns in the token probability sequence may signify different meanings. For instance, sequences of low-probability tokens often imply a lack of model confidence in the semantic correctness of an entire output span. Conversely, if such tokens are scattered, it could be attributed to acoustic noise while the output remains mostly correct. Defining rules manually can be challenging, so we adopt a learning-based approach: constructing a lightweight BiLSTM model to predict the local SU confidence. To achieve this, we label the dev set data into two classes based on their accuracy and the threshold, and then train the BiLSTM model to predict the classes. The model's output serves a similar role as the perplexing score.

CNN-Based Approach The intermediate results from the encoder contain rich information about the data but possess significantly higher dimensions compared to the token probability sequence. To address this, we propose a CNN-based approach to assist in down sampling the intermediate results and predict the confidence. Similar to the RNN-based approach, we train the CNN model on the develop set to predict the local SU accuracy. It's noteworthy that during the design of the CNN network, we observed that the CNN model typically require a larger model size compared to the RNN approach. This issue makes it less practical.

Decision timing The PASU makes the offloading decision precisely when the data ingestion is completed, along with the pilot inference result. This is shown in Figure 3(b).(⑤) Both the perplexing score method and the BiLSTM method require post-decoding results. However, if the system performs confidence estimation after the full local execution, the offloaded data will also suffer from local decoding latency. To circumvent this, the system executes the aforementioned methods on the latest pilot inference result, , i.e. the 3rd pilot inference in Figure 3(b).(②) enabling the system to make the execution path decision simultaneously with the completion of data ingestion. This approach is founded on the assumption that the partial data in the pilot inference can reflect corresponding information in the full-length data.

6 IMPLEMENTATION

We implemented PASU using 4K SLOC in Python, built upon a popular speech toolkit ESPnet [40] .

Model design and training details Our model is motivated by the Branchformer [30], as shown in the comparison in Figure 3(a). Branchformer has parallel weighted sum convolutional and transformer blocks in each layer for local and contextual feature extraction. Our model keeps the blocks that have higher weight and rearrange them to make the convolutional layers at the bottom (near the input). To train our model, we first configure and train an example branchformer model base on the memory and latency budget. In the experiment case, encoder has 9 layers and decoder has 1 layer. To construct PASU's model, assume there are k transformer blocks with weights exceeding $h = 0.2$ in the trained branchformer encoder, we design the encoder with N-k-1 CNN layers followed by transformer k+1 layers. In the experiment, we get 2+1 transformer and 9-2-1 CNN layers. This design is then trained from scratch on the same dataset, with the same decoder design branchformer model has.

Operation details During system operation on the SLURP dataset, we set the shortest partial data for pilot inference at 1.5 seconds. The granularity is determined based on hardware specifications, tested across 0.5s - 2s. The pilot beam size is set to be 3, which is 60% of the full inference beam size of 5. The length limit for pilot inference tokens is set to be 15. The length prediction C is set to be 5.

Hybrid execution details For hybrid execution and offloading decision-making, in the perplexing score-based method, we calculate the perplexing score based on the reference result from the pilot inference. Regarding the LSTM and CNN-based methods, we utilize the full dev set in SLURP as the training set for the LSTM and CNN models. The CNN approach employs a CNN model that takes the encoder's full output as input. The CNN model has two convolutional layers and two fully connected layers, with 192.5 M parameters. The LSTM approach employs a BiLSTM model that takes CTC prefix probability and transformer token-wise probability of the reference result as the input. The LSTM model we adopted is a 1-layer BiLSTM model with 11.1k parameters. The training task is defined as a binary classification task, where a local execution WER > 0.1 is labeled as 1, and otherwise as 0. During operation, the BiLSTM and CNN process the respective input and provide predictions, representing the probability of the data having a WER > 0.1 as a float number between 0 and 1. Further, we set thresholds on this output for different offloading accuracy targets.

Platforms	Compute resources	GFLOPs/s
Jetson Orin Nano	6-core Arm Cortex-A78AE CPU 1.5GHz	14.9
Jetson AGX Xavier	8-core NVIDIA Carmel Arm CPU 2.2GHz	30.8

Table 2: Embedded platforms used in evaluation

	Train set	Dev set	Test set	Overall
# of speakers	167	137	142	177
# of audio files	119880	8690	13078	141648
Total audio len	84.7 h	6.9 h	10.2 h	101.8 h
Audio len / input	2.5±1.1 s	2.9±1.2 s	2.8±1.3 s	2.6±1.1 s
# tokens / input	11.4±6.1	11.5±6.4	11.3±6.7	11.4±6.2

Table 3: Overview of SLURP dataset for SU tasks

7 EVALUATION

We answer the following questions:

- Can PASU reduce latency with competitive accuracy?
- PASU's local path: what is its efficacy?
- PASU's offloading path: what is its efficacy?

7.1 Methodology

Test platforms As shown in Table 2, we test on two embedded boards: Jetson Orin Nano (shortened as Orin) with an efficiency-oriented SoC with six cores, whereas Jetson AGX Xavier (shortened as Xavier) with a performance-oriented SoC with eight cores. Our experiments focus on CPUs, though our idea applies to GPUs as well.

We run the cloud runtime on an x86/NVidia machine and measure accuracy. To better estimate the cloud offloading delay, we measure Microsoft's speech service [28]: we invoke the Azure APIs to offload waveforms and record each waveform's end-to-end wall time. Our measurement runs from the US east coast and invokes datacenters on the east coast. For each input, we repeat the test on enterprise WiFi (bandwidth in 100s of MB and RTT in ms) and on 5G networks, and take the average. Note that we do *not* use Azure speech for accuracy evaluation, as we find its accuracy inferior to the SOTA model used by us (two-pass SLU [1]).

Dataset and model training We run our experiments on SLURP [3] as summarized in Table 3. Comprising 102 hours of speech, SLURP's utterances are complex and closer to daily human speech (e.g. "within the past three months how many meetings did i have with mr richards"), as opposed to short commands (e.g. "light on") in many other datasets. Of all 141,530 utterances, we use 119,880 (85%) for training and the rest for dev set and test set. The accuracy and latency are reported from the test set data range from 2s to 6s. The on-device model training was performed on an RTX 2080 Ti GPU and took 48 hours. It is important to note that this training is a one-time effort.

Metrics We follow the common practice. Accuracy. For ASR, we report word error rate (WER): the discrepancy between the model output and the ground truth transcript, defined as count of errors (substitutions, deletions, insertions) normalized by the sentence length. For SLU, we report the intent classification accuracy (IC).

Latency. (1) We report user-perceive latency [47]: the elapsed time from a user completing her utterance to the system emitting the complete SU result. (2) We further report the real time factor (RTF): the user-perceived latency normalized by the input utterance's duration.

Baselines We compare the following designs.

- *OnDevice*: SU completely runs on device, for which we test a wide selection of model architectures: Conformer [10], Branchformer [30], and CNN-RNNT [48]. Note that CNN-RNNT is streaming, with 640ms chunks as in prior work. For fair comparison, we choose models sizes to be around 30M parameters, which are comparable to ours. They are summarized in Table 4.
- *AllOffload*: The device offloads all inputs, for which the cloud runs a SOTA model [1]: a two-pass end-to-end SLU model with one conformer-based encoder, one conformer-based deliberation encoder, two transformer-based decoders for the two passes, and a pretrained BERT language model. It has >10x parameters as compared to the local models above. We refer to its accuracy as *gold*.
- *NaiveHybrid*: To combine *OnDevice* and *AllOffload*, execute any input with probability α for local execution and $(1-\alpha)$ for offloading.
- *Ours*: By varying the confidence threshold θ, we tested a range of variants which we refer to as *Ours-OnDevice* (0% offloaded), *Ours-Offload-L* (25%), *Ours-Offload-M* (47%), and *Ours-Offload-H* (63%). Here the offloading thresholds and percentages are chosen to meet overall WER targets 0.15, 0.13 and 0.12.

7.2 End-to-end results

As demonstrated in Figure 4, PASU is able to deliver a wide range of accuracies (from modest to nearly gold), with latencies as low as 100s of ms.

Compared to *OnDevice*, all variants of our system are better in the latency - accuracy trade-off. *Ours-Offload-M* and *Ours-Offload-H* achieve much higher accuracy (WER lower by 5%; IC higher by 2.5%) at similar or lower latencies; *Ours-OnDevice* achieves similar accuracies while reducing latencies by around 2x.

Compared to *AllOffload*, *Ours-Offload-L* and *Ours-Offload-M* reduce the overall latency by as much as 37% and 27% while seeing minor degradation in accuracy (no more than 4% WER and 2% of IC). They process 75% and 53% of inputs

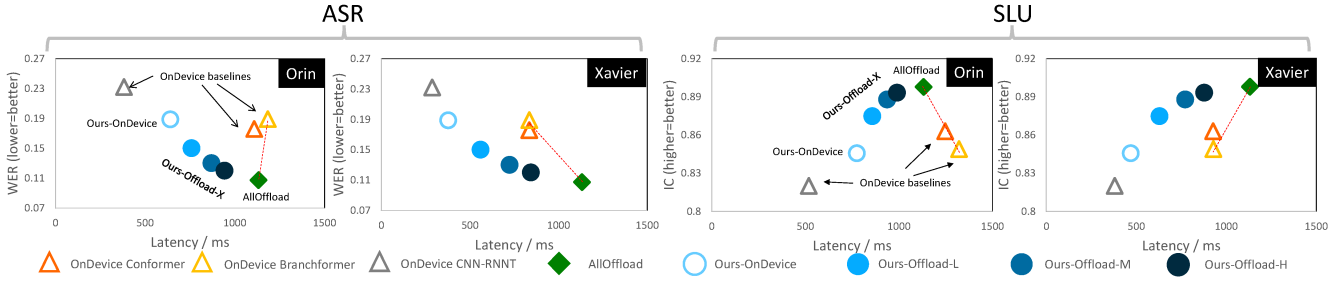

Figure 4: Our system deliver low delays and competitive accuracy. Compared to other local models marked with △△△, our local execution Ours-OnDevice ○ show similar accuracies at much lower delays or higher accuracies; Compared to *AllOffload* ◆, our hybrid executions ● ● ● show much lower delays with little accuracy loss.

	Conformer-M		Branchformer		CNN + RNN-T		Ours	
	Orin	Xavier	Orin	Xavier	Orin	Xavier	Orin	Xavier
Params	27.21M		36.02M		24.18M		23.33M	
Encode. embed	0.76G		3.02G		3.02G		3.02G	
	0.01	0.01	0.04	0.02	0.03	0.02	0.03	0.02
Encode. encoders	1.54G		1.88G		1.00G		1.00G /0.34G	
	0.10	0.09	0.14	0.12	0.10/0.02	0.10/0.02	0.08/0.02	0.07/0.02
Encode. Total	2.30G /1.54G		4.90G/1.88G		4.02G		4.02G/0.34G	
	0.11/0.10	0.10/0.09	0.17/0.14	0.15/0.12	0.13/0.02	0.12/0.02	0.11/0.02	0.10/0.02
Decode	0.26	0.17	0.24	0.14	0.50/0.10	0.35/0.07	0.18	0.10

Underscore = the portion of workloads that cannot be processed in streaming

Table 4: Our local encoder executes mostly streaming FLOPs, for which latency can be hidden behind IO; doing so does not compromise the final accuracy. FLOPs numbers normalized to 1 second of input. Our local execution path incurs lower latencies (reported as RTF) as compared to other on-device models.

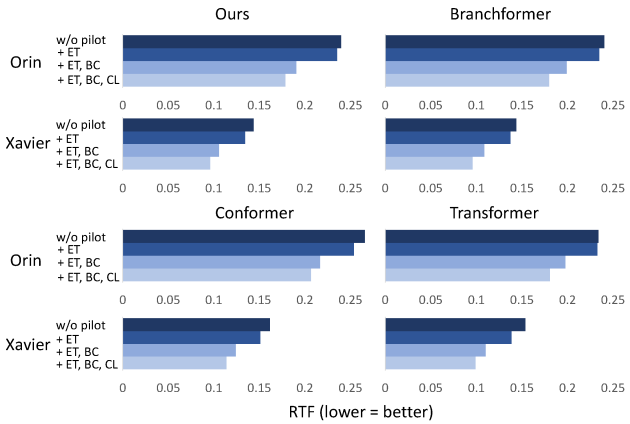

Figure 5: An ablation study of *pilot inference*, showing that all its three optimizations contribute to lower latency significantly. It also shows that pilot inference has generalizability to be applied to various on-device models (e.g. Branchformer, Conformer, and Transformer) other than ours. ET: Early termination, BC: Beam collapse, CL: CTC Leap

on device, reducing the need for cloud invocations by 2x or more. *Ours-Offload-H* has negligible accuracy degradation (<0.5% of IC) while still reducing latency by 20% and offload needs by 37%.

NaiveHybrid, shown as the dash lines interpolating between *OnDevice* and *AllOffload* on Figure 4, always shows inferior accuracy or latency to ours.

7.3 Efficacy of our local path

As shown in Figure 4, our local-only execution (*Ours-OnDevice*) significantly outperforms *OnDevice*, reducing latencies by 1.7x - 2.2x at similar accuracies. We next break down the benefit into two parts: encoding and decoding.

Encoding speedup We strike a sweet spot between speed (by processing inputs in a streaming fashion) and accuracy (by attending to the whole input). Table 4 breaks down the encoding computation. Compared to encoders with all attention layers (Con/Branchformers) where 67% and 38% FLOPs are non-streaming (i.e. must execute *after* ingestion), 92% FLOPs of ours is streaming, for which the latency is hidden behind the ingestion. Meanwhile, we do not lose much accuracy compared to Con/Branchformers, as our design runs three attention layers after ingestion ends. This design reduces the latency by 3.6x - 5.4x (220 – 345 ms or 0.07 – 0.11 RTF) as shown in Table 4. Compared to full streaming encoders (CNN+RNNT) for which 100% FLOPs is streaming, our accuracy is much higher by 4% in WER and 2.6% in IC, because the former critically lacks attention over the whole input. Meanwhile, our encoding latency is only higher by 0.005 RTF (15 ms for a 3 second input) as in Table 4. Such a difference is dwarfed by the difference in decoding delays.

Decoding speedup Our design shows 1.5x lower decoding delays compares to *OnDevice* (Table 4). All three techniques show contribution to the overall speedup as Figure 5 shows. We also demonstrate the decoding speedup method on branchformer. The speedup ratio is similar with it on ours model, shows that our speedup method is applicable on general transformer based encoder-decoder speech model.

		OnDevice	Ours-Offload-L		Ours-Offload-H	
		RTF	RTF	Tgt / nTgt	RTF	Tgt / nTgt
w/o pilot inf		0.168	0.238	1370 / 336	0.356	3306 / 1267
w/ pilot inf	τ=2s	0.145	0.209	1743 / 932	0.298	3665 / 2906
	τ=1s	0.138	0.197	1653 / 826	0.286	3538 / 2637
	τ=0.5s	0.120	0.178	1644 / 657	0.268	3534 / 2264

Table 5: Pilot inference reduces the end-to-end delay. It benefits our local-only execution (Ours-NoOffload) as well as our hybrid executions (Ours-Offload-X), making the latter's offloading more selective. Tgt / nTgt refers to target / non-target inputs, respectively, in which Tgt are the input data on which the cloud gets lower WER, i.e. they should be offloaded.

		WER=0.15		WER=0.14		WER=0.13		WER=0.12	
		RTF	offload	RTF	offload	RTF	offload	RTF	offload
NaiveHybrid		0.234	47.6%	0.264	59.8%	0.293	72.2%	0.323	84.5%
DeeBERT		-	-	-	-	-	-	-	-
Ours	CNN	0.190	27.6%	0.215	37.8%	0.243	49.8%	0.280	65.7%
	LSTM	0.171	21.5%	0.195	31.1%	0.222	43.0%	0.263	60.2%
	PPL	0.178	25.0%	0.200	34.5%	0.230	47.3%	0.268	63.1%

Table 6: Our offloading strategies based on sequence modeling outperform *NaiveHybrid* (random selection) and DeeBERT (sequence classification, failing to produce useful results). In the experiment, we tune θ and α to meet WER goals and compare delays and offloading ratio

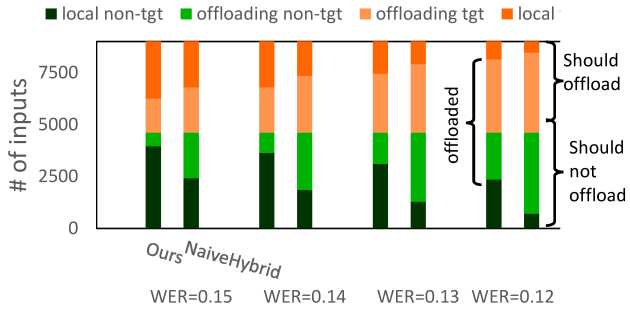

Figure 6: Our design shows good selectivity in making offloading decisions, offloading much fewer inputs compared to *NaiveHybrid* achieving similar accuracies. tgt / non-target refer to target / non-target input, target input is data that get lower WER on cloud

The *incremental* nature of pilot inference is crucial, as it amortizes the decoding cost over the past input, making the cost scale slower as the input grows. Turning off the incremental design (i.e. each pilot inference starts from the input's start) increases the average decoding delay by 30%, from 156 ms to 205 ms; it increases the 90th percentile delay by 40%, from 198 ms to 282 ms. Reduction in pilot inference cost allows the system to execute pilot inference more frequently. The pilot inference shows generalizability on mainstream SOTA speech models. As Figure 5 shows, besides our model, the pilot inference can effectively bring speedup to Branchformer, Conformer (with transformer decoder) and Transformer model. All three techniques contribute to the overall latency reduction.

We further study the impact of pilot inference's eagerness: during ingestion, every τ seconds PASU decodes the input it has accumulated so far. Lower τ reduces the discrepancy between the last pilot inference and the full decoding, therefore improving the full decoding speed and quality. As shown in Table 5, 4x reduction in τ (from 2s to 0.5 s) reduces final RTF by 0.025. further reduction in τ helps the pilot inference get longer partial data that cover more part of the full length data In exchange, the expense is that the ingestion consumes more compute. τ is lower bounded by the available compute resource during ingestion. For instance, Jetson AGX Xavier can sustain $\tau = 0.5s$ while Jetson Orin Nano cannot.

7.4 Efficacy of our offloading path

PASU effectively identifies and uploads the inputs that would suffer from low accuracy on device. With our selective offloading technique, PASU achieves lower latency and offloading cost. In this section, we define data as target inputs (shortened as Tgt) when they get lower WER (i.e. higher accuracy) from the cloud processing than local processing.

Comparison vs. *NaiveHybrid* We replace PASU's offloading strategy with *NaiveHybrid* while keeping all other optimizations. The results in Table 6 show that to reach the same accuracy (WER), *NaiveHybrid* has to offload up to 2x more inputs. The extra offloading translates to higher cloud cost as well as higher latency (0.06 RTF; 180 ms on average).

Figure 6 shows detailed offloading decisions: For *Ours-Offload-M* (WER=0.13) and *Ours-Offload-H* (WER=0.12), PASU offloads the majority of target inputs, while still executing the majority of non-target inputs on device. This is much higher than *NaiveHybrid* which make decisions "by chance".

Comparison vs. DeeBERT Our experiment also shows that a popular early-exit approach [43], estimating model confidence with linear classifiers inserted after the encoder, performs poorly for encoder-decoder SU models. We implemented such an approach: training linear classifiers atop the first output frame embedding from the encoder. We could not get meaningful results – the classifier is no better than a random predictor. This highlights the challenge of predicting

SU confidence, for which the entire generated sequence (not just the 1st frame from encoding) must be considered.

Choice of sequence modeling strategies The sequence modeling techniques in Section 6 (CNN, LSTM, and perplexity) can well estimate the model confidence. Table 6 shows that LSTM offers the lowest offloading percentage and RTF. PPL perform slightly worse, but as perplexity has much lower computational complexity (no training, no parameters), we deem PPL as the most suitable.

How PI affects offloading decisions? Recall that PASU makes offloading decisions based on the last pilot inference (section 5). We answer the following questions. (1) How does pilot inference's execution plan affect offloading decisions? Our results show that finer granularity (i.e. smaller discrepancy between the pilot and the full inference) leads to better estimation of model confidence, which results in more accurate offloading decisions. For instance in Table 5, to maintain the accuracy at WER=0.15, reducing τ from 2 s to 0.5 s reduces the offloaded inputs by 14%, which translates to 93 ms lower end-to-end delay. (2) What if we make offloading decisions based on the *full* decoding outcome? Our results show that while the offloading selectivity slightly improves, the end-to-end latency is much higher (by 264 ms or 0.088 RTF), because the device makes offloading decisions much late – after ingesting the whole input and executing full decoding.

7.5 Energy impact

Methodology We study system-level energy consumption to demonstrate the impact of our techniques, specifically focusing on the overall energy consumption during speech processing. The experiments are conducted on Jetson AGX Xavier with the built-in power profiling tool tegrastats. Power profiling is performed during inference on all test data, with a measurement interval of 20ms to capture the device's overall power consumption. Six different settings are tested: four on PASU, include *OnDevice*, *Ours-Offload-L*, *Ours-Offload-M*, and *Ours-Offload-H*. The remaining two settings involve the default full encoding-decoding execution scheme with our on-device model and the branchformer model. The average energy consumption on each voice input is shown in Figure 7, with a detailed breakdown of energy consumption across three execution stages: data ingestion, pilot inference (if applicable), and full inference.

Energy comparison In the experiments mentioned above, we observed a 32% energy overhead on our *OnDevice* system compared to the two default full encoding-decoding systems with ours on-device model and the branchformer model. The average inference energy consumption breakdown is shown in Figure 7. From the figure, we can see that although the pilot inference introduces energy overhead, the full inference is shorter, therefore reducing the overall energy consumption overhead. With the selective offloading, PASU can further reduce the energy consumption overhead by up to 44%. It's important to note that the Jetson AGX Xavier used in the experiments is relatively power inefficient. We assume that more efficient modern chips, such as Apple M1 and A17, would result in a lower energy overhead.

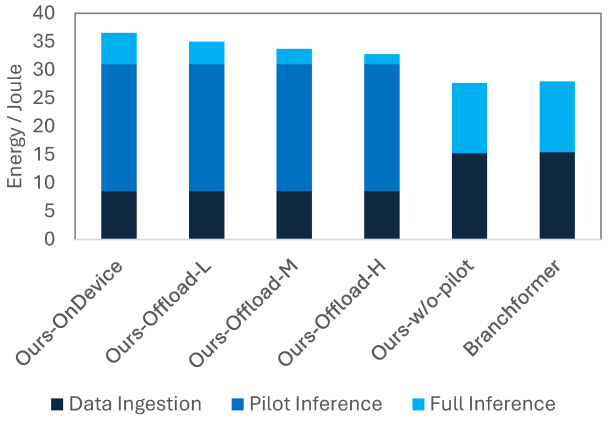

Figure 7: Our system incurs minor energy overhead (18% to 32%) as compared to the baseline system.

What-if analysis Many factors may influence the overall energy consumption. Assuming we have larger models, PASU will see greater latency reduction, which therefore saves more energy in the full inference process. Designing a more energy-efficient model can also help reduce the overhead of PASU in the pilot inference stage. Also, as mentioned earlier, more efficient hardware processors can help mitigate the energy overhead of pilot inference.

7.6 Discussion

Device hardware As mentioned in Section 3, our system targets commodity devices capable of executing the local model notably faster (e.g. >50%) than offloading to the cloud, in order to warrant the higher local WER (often by 7%-10%). Less capable devices, however, experience longer latency, making it impractical compared to simply offloading to the cloud. The problems are twofold: constrained hardware leads to longer pilot inference latency and makes the intervals between consecutive pilot inferences longer, impairing its efficacy for both local and offloading paths. Additionally, the resource constraints also slow down the full decoding. Experiments on a Raspberry Pi 4b with the same model and system settings showed that the pilot inference intervals cannot be less than 1.2 seconds, otherwise some of the pilot inference could not finish before the next pilot inference starts. The overall latency was 0.343 RTF, only marginally

faster than cloud offloading. While our system can further compress the model to suit the device resource constraints, this would also result in a drop in accuracy.

Within the capable device regime, our local execution maintains consistent speed advantages over baselines, as shown by the comparison Jetson Orin Nano and Jetson AGX Xavier in Figure 4. Besides, faster device computation can perform pilot inference with smaller interval.

Network conditions Our measurement was from decent network conditions that favor offloading. With longer network delays, PASU's benefit would be even more pronounced, as fast local executions and selective offloading will be more important. If network delays further reduce, the latency gap between our system and *OffloadAll* may narrow. Even if that happens, our benefit of reduced cloud cost will still remain. Network conditions, especially when unstable, could also be a key consideration in offloading decisions. The system may need to adjust the offloading threshold based on network latency and availability. Under typical network conditions, placing the SU model on a nearby edge system may not improve overall performance. This is because the SU model on the cloud is resource-intensive, running it on the edge could result in a latency increase that is significantly longer than the network roundtrip latency.

8 RELATED WORK

LLM speculative decoding (SD) Speculative decoding for LLM decoding speedup includes a fast generating and verify later pattern. Examples include copy-and-verify scheme [45], non-autoregressive translation methods [42], shallow decoder and parallel decoding [38], token confidence based decoding [15]. Similar techniques have been adopted in speech processing, such as the Whisper Speculative Decoding [7] in which the audio is first encoded with a transformer encoder, then decoded by a distilled, lightweight decoder initially, and later verified by a larger decoder through a one-pass forward process. **Comparing SD to our pilot inference** *Tasks:* Pilot inference targets streaming SU task, addressing temporal workload imbalance. SD targets LLM and SU decoding, tackling underutilized hardware parallelism. *Solutions:* Pilot inference work with incomplete streaming input. SD deals with a complete input with concurrent fast and slow tasks. SD's speedup is contingent upon high hardware parallelism, which is lacking on embedded devices. The connection in between is that the full decoding benefit from an earlier decoding process. More specifically, Whisper Speculative Decoding still encodes the complete audio and decodes with two-pass speculative scheme all after the ingestion finished. In contrast, our system exploits the temporal load imbalance, finishs part of the encoding and pilot inference during the audio ingestion and saves a substantial amount of time.

Streaming speech procssing Streaming approaches like transducer-based models [9, 35, 48] process the data in a chunk-wise fashion, initiating transcription during data ingestion and reduces the latency. However, they only possess partial data during streaming processing and lack contextual information. Compare to attention-based model [2, 10, 30], the accuracy of streaming models is inferior [20]. Our work focuses on speedup attention-based models.

Offloading and hybrid execution Device-cloud collaboration [23] is well-discussed in general scenarios. The cost during collaboration is an important concern [17, 27, 33]. A related work optimizes SU for microcontrollers in the local/cloud setting [4]. Yet, incapable of deep SU models , microcontrollers only run an inference *cache*, rather than a complete engine like PASU. These two projects have orthogonal contributions; they do not depend each other.

Comparing CTC Leap with the Online CTC/Attention Method Online CTC/attention method [26] reduces CTC scoring latency by Truncated CTC method. Both Truncated CTC and CTC leap try to skip part of the CTC prefix scoring computation among all the time frame. The difference is that: we reuse the CTC computation from the pilot inference to skip the beginning part; whereas they stop early by using the CTC peak heuristics to skip the latter part. Notably, same with the original approach [41] our method still includes all the time frames while they choose to drop some temporal information, which may include crucial information for CTC.

Mobile speech processing Various efforts have been made to run SU tasks on resource-constrained mobile devices. LUT-NN [39] employed table lookup inference execution for machine learning models to enhance inference speed and reduce memory footprint. However, the lookup table scheme is specifically designed for the encoding process and cannot be easily applied to generative tasks in SU. Other methods such as quantization [8, 44] and pruning [16, 31] have also been proposed. These methods are orthogonal to our approach.

9 CONCLUSIONS

We present PASU, a novel system for fast speech processing in SU tasks. PASU contributes late contextualization, beam collapse/ termination, and CTC leap for local execution; and confidence estimation for selective offloading. All optimizations combined, PASU speeds up its local path by 2x and reduces the offloading needs by 2x in its offloading path.

ACKNOWLEDGMENT

The authors were supported in part by NSF awards #2128725, #1919197, #2106893, and Virginia's Commonwealth Cyber Initiative. The authors thank the anonymous reviewers for their insightful feedback.

REFERENCES

[1] Siddhant Arora, Siddharth Dalmia, Xuankai Chang, Brian Yan, Alan Black, and Shinji Watanabe. Two-pass low latency end-to-end spoken language understanding. In *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, volume 2022, pages 3478–3482, 2022.

[2] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. *Advances in neural information processing systems*, 33:12449–12460, 2020.

[3] Emanuele Bastianelli, Andrea Vanzo, Pawel Swietojanski, and Verena Rieser. SLURP: A Spoken Language Understanding Resource Package. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

[4] Afsara Benazir, Zhiming Xu, and Felix Xiaozhu Lin. Leveraging cache to enable slu on tiny devices, 2023.

[5] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[6] Renato De Mori. Spoken language understanding: A survey. In *2007 IEEE Workshop on Automatic Speech Recognition & Understanding (ASRU)*, pages 365–376. IEEE, 2007.

[7] Sanchit Gandhi. Speculative decoding for 2x faster whisper inference, 2023. Accessed: 2024-7-21.

[8] Santosh Gondi. Wav2vec2.0 on the edge: Performance evaluation, 2022.

[9] Alex Graves. Sequence transduction with recurrent neural networks. *arXiv preprint arXiv:1211.3711*, 2012.

[10] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, and Ruoming Pang. Conformer: Convolution-augmented transformer for speech recognition. *CoRR*, abs/2005.08100, 2020.

[11] Awni Y. Hannun. The history of speech recognition to the year 2030. *ArXiv*, abs/2108.00084, 2021.

[12] Wei-Ning Hsu, Benjamin Bolte, Yao-Hung Hubert Tsai, Kushal Lakhotia, Ruslan Salakhutdinov, and Abdelrahman Mohamed. Hubert: Self-supervised speech representation learning by masked prediction of hidden units. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 29:3451–3460, oct 2021.

[13] Frederick Jelinek, Robert L. Mercer, Lalit R. Bahl, and Janet M. Baker. Perplexity—a measure of the difficulty of speech recognition tasks. *Journal of the Acoustical Society of America*, 62, 1977.

[14] Jungo Kasai, Keisuke Sakaguchi, Ronan Le Bras, Dragomir Radev, Yejin Choi, and Noah A. Smith. Beam decoding with controlled patience, 2022.

[15] Sehoon Kim, Karttikeya Mangalam, Suhong Moon, John Canny, Jitendra Malik, Michael W. Mahoney, Amir Gholami, and Kurt Keutzer. Speculative decoding with big little decoder. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2023.

[16] Cheng-I Jeff Lai, Yang Zhang, Alexander H Liu, Shiyu Chang, Yi-Lun Liao, Yung-Sung Chuang, Kaizhi Qian, Sameer Khurana, David Cox, and Jim Glass. Parp: Prune, adjust and re-prune for self-supervised speech recognition. *Advances in Neural Information Processing Systems*, 34:21256–21272, 2021.

[17] Pat Lawlor. Generative ai trends by the numbers: Costs, resources, parameters and more, 2023. Accessed: 2023-7-26.

[18] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 873–887. USENIX Association, July 2020.

[19] Soyoon Lee and Hyokyung Bahn. Characterization of android memory references and implication to hybrid memory management. *IEEE Access*, 9:60997–61009, 2021.

[20] Jinyu Li, Yu Wu, Yashesh Gaur, Chengyi Wang, Rui Zhao, and Shujie Liu. On the comparison of popular end-to-end models for large scale speech recognition. In Helen Meng, Bo Xu, and Thomas Fang Zheng, editors, *Interspeech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25-29 October 2020*, pages 1–5. ISCA, 2020.

[21] Baiyang Liu, Björn Hoffmeister, and Ariya Rastrow. Accurate endpointing with expected pause duration. In *Interspeech*, 2015.

[22] Chengfei Lv, Chaoyue Niu, Renjie Gu, Xiaotang Jiang, Zhaode Wang, Bin Liu, Ziqi Wu, Qiulin Yao, Congyu Huang, Panos Huang, et al. Walle: An {End-to-End},{General-Purpose}, and {Large-Scale} production system for {Device-Cloud} collaborative machine learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 249–265, 2022.

[23] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656, 2017.

[24] Sumit Maheshwari, Dipankar Raychaudhuri, Ivan Seskar, and Francesco Bronzino. Scalability and performance evaluation of edge cloud systems for latency constrained applications. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 286–299, 2018.

[25] A. Martin, D. Charlet, and L. Mauuary. Robust speech/non-speech detection using lda applied to mfcc. In *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)*, volume 1, pages 237–240 vol.1, 2001.

[26] Haoran Miao, Gaofeng Cheng, Pengyuan Zhang, and Yonghong Yan. Online hybrid ctc/attention end-to-end automatic speech recognition architecture. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 28:1452–1465, 2020.

[27] microsoft. Azure hybrid benefit, 2023. Accessed: 2023-11-18.

[28] Brian Mouncer, Henry van der Vegte, and Mark Hillebrand. Azure-samples/cognitive-services-speech-sdk, 2023. Accessed: 2023-9-24.

[29] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, 2015.

[30] Yifan Peng, Siddharth Dalmia, Ian Lane, and Shinji Watanabe. Branchformer: Parallel mlp-attention architectures to capture local and global context for speech recognition and understanding. In *International Conference on Machine Learning*, pages 17627–17643. PMLR, 2022.

[31] Yifan Peng, Kwangyoun Kim, Felix Wu, Prashant Sridhar, and Shinji Watanabe. Structured pruning of self-supervised pre-trained models for speech recognition and understanding. In *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5, 2023.

[32] Golan Pundak, Tara N. Sainath, Rohit Prabhavalkar, Anjuli Kannan, and Ding Zhao. Deep context: End-to-end contextual speech recognition. In *2018 IEEE Spoken Language Technology Workshop (SLT)*, pages 418–425, 2018.

[33] qualcomm. The future of ai is hybrid, 2023. Accessed: 2023-11-18.

[34] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine Mcleavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 28492–28518. PMLR, 23–29 Jul 2023.

[35] Kanishka Rao, Haşim Sak, and Rohit Prabhavalkar. Exploring architectures, data and units for streaming end-to-end speech recognition with rnn-transducer. In *2017 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 193–199. IEEE, 2017.

[36] Alaa Saade, Joseph Dureau, David Leroy, Francesco Caltagirone, Alice Coucke, Adrien Ball, Clément Doumouro, Thibaut Lavril, Alexandre Caulier, Théodore Bluche, et al. Spoken language understanding on the edge. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 57–61. IEEE, 2019.

[37] Ali Shakarami, Mostafa Ghobaei-Arani, and Ali Shahidinejad. A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective. *Computer Networks*, 182:107496, 2020.

[38] Xin Sun, Tao Ge, Furu Wei, and Houfeng Wang. Instantaneous grammatical error correction with shallow aggressive decoding. *arXiv preprint arXiv:2106.04970*, 2021.

[39] Xiaohu Tang, Yang Wang, Ting Cao, Li Lyna Zhang, Qi Chen, Deng Cai, Yunxin Liu, and Mao Yang. Lut-nn: Empower efficient neural network inference with centroid learning and table lookup. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom '23, New York, NY, USA, 2023. Association for Computing Machinery.

[40] Shinji Watanabe, Takaaki Hori, Shigeki Karita, Tomoki Hayashi, Jiro Nishitoba, Yuya Unno, Nelson Enrique Yalta Soplin, Jahn Heymann, Matthew Wiesner, Nanxin Chen, Adithya Renduchintala, and Tsubasa Ochiai. ESPnet: End-to-end speech processing toolkit. In *Proceedings of Interspeech*, pages 2207–2211, 2018.

[41] Shinji Watanabe, Takaaki Hori, Suyoun Kim, John R. Hershey, and Tomoki Hayashi. Hybrid ctc/attention architecture for end-to-end speech recognition. *IEEE Journal of Selected Topics in Signal Processing*, 11(8):1240–1253, 2017.

[42] Heming Xia, Tao Ge, Furu Wei, and Zhifang Sui. Lossless speedup of autoregressive translation with generalized aggressive decoding. *arXiv preprint arXiv:2203.16487*, 2022.

[43] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. DeeBERT: Dynamic early exiting for accelerating BERT inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2246–2251, Online, July 2020. Association for Computational Linguistics.

[44] Junhao Xu, Shoukang Hu, Jianwei Yu, Xunying Liu, and Helen Meng. Mixed precision quantization of transformer language models for speech recognition. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7383–7387, 2021.

[45] 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.

[46] Shuochao Yao, Jinyang Li, Dongxin Liu, Tianshi Wang, Shengzhong Liu, Huajie Shao, and Tarek Abdelzaher. Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, page 476–488, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Dong Yu and Lin Deng. *Automatic speech recognition*, volume 1. Springer, 2016.

[48] Qian Zhang, Han Lu, Hasim Sak, Anshuman Tripathi, Erik McDermott, Stephen Koo, and Shankar Kumar. Transformer transducer: A streamable speech recognition model with transformer encoders and rnn-t loss. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7829–7833. IEEE, 2020.

[49] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. Bert loses patience: Fast and robust inference with early exit. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 18330–18341. Curran Associates, Inc., 2020.