CHA: A Caching Framework for Home-based Voice Assistant Systems

Lanyu Xu
Wayne State University
Detroit, MI, 48202
xu.lanyu@wayne.edu

Arun Iyengar
IBM T.J. Watson Research Center
Yorktown Heights, NY, 10598
aruni@us.ibm.com

Weisong Shi
Wayne State University
Detroit, MI, 48202
weisong@wayne.edu

***Abstract*—Voice assistant systems are becoming immersive in our daily lives nowadays. However, current voice assistant systems rely on the cloud for command understanding and fulfillment, resulting in unstable performance and unnecessary frequent network transmission. In this paper, we introduce *CHA*, an edge-based caching framework for voice assistant systems, and especially for smart homes where resource-restricted edge devices can be deployed. Located between the voice assistant device and the cloud, *CHA* introduces a layered architecture with modular design in each layer. By introducing an understanding module and adaptive learning, *CHA* understands the user's intent with high accuracy. By maintaining a cache, *CHA* reduces the interaction with the cloud and provides fast and stable responses in a smart home. Targeting on resource-constrained edge devices, *CHA* uses joint classification and model pruning on a pre-trained language model to achieve performance and system efficiency. We compare *CHA* to the status quo solution of voice assistant systems and show that *CHA* benefits voice assistant systems. We evaluate *CHA* on three edge devices that differ in hardware configuration and demonstrate its ability to meet the latency and accuracy demands with efficient resource utilization. Our evaluation shows that compared to the current solution for voice assistant systems, *CHA* can provide at least 70% speedup in responses for frequently asked voice commands with less than 13% CPU consumption, and less than 9% memory consumption when running on a Raspberry Pi.**

***Index Terms*—Edge Computing; Voice Assistant Systems; Caching**

I. Introduction

The widespread development of computerized natural language understanding (NLU) has greatly affected how people interact with machines. Smart Speakers, such as the Amazon Echo, Google Home have been benefited from NLU technology and are entering millions of families across the world. In 2019, nearly 150 million smart speakers were sold across the world, which is 70% higher than 2018 [1]. A survey from Voicebot reported that nearly 90 million U.S. adults have smart speakers in January 2020, which increased by 85% in two years [2]. Not only the smart speaker, but there are also more and more open-sourced smart home systems that are taking voice control into the development. For example, HomeAssistant [3] introduced Ada [4] as its voice assistant several months ago, and openHAB [5] integrated Google Assistant to provide voice control function. The burgeoning voice assistant market encourages a variety of smart devices to integrate with voice interaction.

As a unique member in a smart home, the voice assistant plays a dual role: on one side, it takes the speech command to the system as a controller for the user to control home devices without a touch; on the other, it reacts to the command as a controlled device.

The smart speaker's pervasive existence has brought intelligence to the home environment and broadened home life. However, it is noticeable that currently, the smart speakers and other voice assistants rely on the cloud for command understanding and fulfillment [6]–[10]. That means if a network outage happens, there is no way for a householder to use the voice assistant. Moreover, even when the network is available, users usually suffer from variations in response latency due to the unpredictable situation during data transmission on the Internet. However, it is not necessary to rely on the cloud to process all the commands. For example, home automation commands can benefit from history fulfillments; task management commands do not have rely on cloud resources. In this paper, we propose an edge-based caching framework that provides support to accelerate the voice assistant system's performance, and the goal is to provide instant-response and high-accuracy service.

In addition, introducing edge-based processing to the current workflow also offers higher reliability, especially when the network is unstable or cloud servers are not responsive. The local home network can process the command to manipulate automation devices in the home environment. User privacy and security also benefit from this computing paradigm, with less data is exposed to the external network.

The voice assistant system relies on intelligent technology to understand voice commands. Cloud-based processing can provide computation and storage resources to handle services, with a trade-off between response latency and performance. In recent years, hardware has been rapidly improving in performance [11], making it possible to realize intelligence at the edge. Compared to the cloud, the computation resources are relatively restricted at the edge. Many efforts have been devoted to enabling edge intelligence [12], [13], in both hardware (*e.g.,* Intel neural compute stick, Nvidia Jetson series, Google edge TPU) and software (*e.g.,* ROS, TensorFlow Lite).

We are inspired by the promotion of edge intelligence to explore edge-supported voice assistant systems. There are several open challenges brought from migrating the command

understanding from the cloud to the edge. The first is how to ensure low response latency since the intelligent model is introduced to the edge processing. By offloading the command understanding to the edge, the latency caused by the unstable network can be alleviated, while the computation limited edge resources may take a longer time to process intelligent tasks.

A second point is whether the performance will degrade from cloud to edge. The cloud-based solution is a trade-off between latency and accuracy. When introducing the understanding function to edge servers with limited computational resources, the understanding performance may be affected because the original large model needs to be compressed to accommodate the edge resources.

A third point is the generality of the edge-supported solution. With the popularity of voice assistant systems and different available open-source smart home systems, the platforms that piggyback voice assistants have a variety of differences in terms of architecture, memory, GPU, *etc.*. It is worth exploring the generality or restrictions of the edge-supported voice command processing in these systems.

In order to explore and address the above-described challenges, we propose *CHA*, an edge-supported caching framework for home-based voice assistant systems. *CHA* is composed by four layers: command interface layer, command parser layer, command caching layer, and management layer. *CHA* implements several optimizations to improve system performance. To reduce latency, *CHA* combines two classification tasks and jointly parses them. To increase accuracy, *CHA* takes feedback from the user to continuously fine-tune the understanding model. To improve the system efficiency, *CHA* provides a lightweight understanding module which performs model compression without hurting the accuracy.

We implement *CHA* in Python and integrate it with speech command understanding models. While the command parser layer parses the assigned command, we also trained a cloud-based command understanding model on Google Dialogflow [6] as the cloud support to the system. It receives commands rerouted to the cloud by the rerouting module and responsible for the integration fulfillment.

We evaluate *CHA* using a spoken language understanding (SLU) dataset across three different hardware platforms and demonstrate that *CHA* can provide general edge-support for voice assistant systems with low latency, high accuracy, and acceptable system efficiency. We compare *CHA* with a pure cloud-based solution and show that *CHA* is a promising solution for the voice assistant systems.

We summarize the contribution of this paper as follows:

- We analyze the cloud-based voice assistant system and find two drawbacks affecting user experience: long latency and unstable response.
- We propose and implement *CHA*, an edge-based caching framework for voice assistant systems to address the two drawbacks. *CHA* integrates command understanding and caching to enable home-based command processing.
- We take a set of techniques, including joint classification and model pruning on a pre-trained understanding model to achieve efficiency performance for *CHA* running on resource-constraint edge devices, and validate its performance and system efficiency on Raspberry Pi.

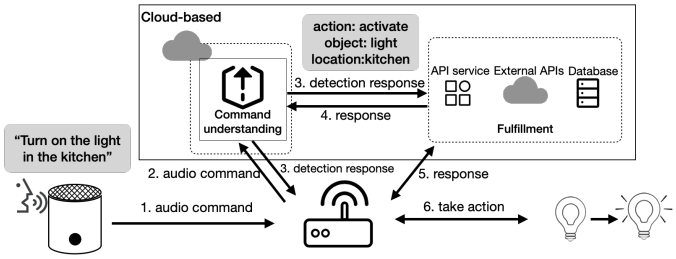


Fig. 1: Status quo: cloud-only processing.

The rest of the paper is organized as follows. We introduce the problem statement and challenges in Section II. Section III describes the design of *CHA*. Its implementation and evaluation are presented in Section IV and V respectively. We discuss the observations about the edge-based efficient voice processing in Section VI. Section VII studies the related work, and Section VIII concludes the paper.

II. Problem Statement and Challenges

Voice assistant systems are widely used. Currently, the command understanding and fulfillment are entirely based on the cloud. In this section, we first analyze the cloud-based voice assistant system in terms of its latency and availability. We then propose the edge-supported processing as a promising solution to address the drawbacks underlying cloud-based solutions. Finally, we discuss the key challenges in edge-supported voice command processing and *CHA*'s solution.

A. The Status Quo: Cloud-Only Processing

First, we explain voice assistant workflow and describe how a voice command is parsed and fulfilled.

As shown in Figure 1, in a smart home environment where the voice assistant has been set up with access to other smart devices, a user asks the voice assistant to turn on the kitchen light by saying "Turn on the light in the kitchen" after awaking the voice assistant with a specified hotword. The voice assistant then sends the audio command to the cloud for processing. The goal for the command understanding module is to understand the command by detecting the *intent* and filling the *slots*. To this end, a pre-trained natural language understanding (NLU) model is stored in the cloud for parsing natural language. Thus, a voice command needs to go through an automatic speech recognition (ASR) model to generate corresponding text from the audio wave before feeding into the NLU model. After that, the intent (*kitchen light on*) is detected, and three slots {*action: activate, object: light, location: kitchen*} are filled. To clarify, in the rest of the paper, we name this two-step parsing model as ASR-NLU model.

The detected intent and slot will then be sent to the fulfillment step. As a smart device control command, the home gateway takes the response back and interacts with the light management module to trigger the light in the kitchen. For

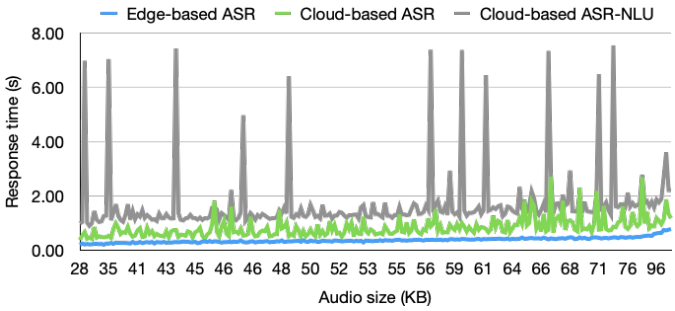


Fig. 2: Voice processing latency comparison.

other types of commands, for example, the "get weather" command, the gateway or the cloud will interact with third-party APIs or the cloud for information, depending on the integration configuration.

Observed from our usage of voice assistants, as well as the complaint from the community [14], [15], the slow response of voice assistants happens from time to time. When processing common commands such as open the light in certain areas, sometimes it takes considerably more time than usual time to respond, even in a stable network connection.

To quantitatively describe the user experience, we analyze the performance of cloud-based voice assistant systems on the voice command recognition and voice command understanding parts, respectively. The voice command recognition is provided by Google Speech-To-Text API [16]. The voice command understanding is provided by Google Dialogflow. These two cloud services provide backend support for products integrating conversational user interface, and represent the state-of-the-art performance. In addition, we deploy an edge-based ASR model on a Raspberry Pi as a comparison. The cloud response time is measured in the round trip time.

Figure 2 demonstrates the response time consumed when processing different sizes of audio data on the edge and the cloud. The audio data contains the different lengths of spoken words from 1 to 9, and the audio size ranges from 20 KB to 133 KB. With audio size increases, the response time for ASR processing increases accordingly, while the cloud-based ASR shows more fluctuation comparing to the edge-based processing. Compared to ASR processing, the cloud-based ASR-NLU exhibits more unstable response latency.

On average, edge-based ASR processing takes 36.9 milliseconds, 53.2% faster than the cloud-based ASR. Table I lists the performance of ASR on cloud and edge in terms of average word error rate (WER) and sentence accuracy. WER is the ratio that word is mistakenly recognized, and lower WER means better performance of the ASR model. The cloud model is a general solution that provides state-of-the-art accuracy, and it achieves 83.2% in sentence accuracy, with 10.42% WER. The edge ASR model is trained with a small while dedicated language vocabulary dictionary, and outperforms the general model with 96.1% accuracy, 2.5% WER. The resource-limited edge shows the capability to perform intelligence with low latency and high accuracy with a dedicated trained model. Moreover, it brings a more stable experience with less interaction with the external network.

	Word error rate (WER)	Sentence accuracy
Cloud-based ASR	10.42%	83.19%
Edge-based ASR	2.52%	96.12%

TABLE I: Voice recognition performance comparison.

[Motivation 1] The cloud-based voice assistant system experiences long latency, unstable performance in response, while an edge-based device can eliminate these drawbacks. Thus the current cloud-based solution can involve the edge to support the voice command fulfillment.

B. The Usage Pattern of Voice Assistant

Next, we review the usage pattern of the home-based voice assistant systems.

Home automation has been developed for several years as one practical implementation of the smart home. Different commercialized or open-sourced products are available in this community. Correspondingly, home devices, including illumination systems, entertainment systems, surveillance systems, and more, are touchless controllable through a voice assistant. The smart home system provides a user interface for the user to send voice commands and fulfill the command accordingly to realize automation. Smart home systems have a dedicated configuration of each device to identify and define the fulfillment actions. A device need to be registered with $\{trigger,\ action\}$ in the configuration file for the system to recognize and fulfill. To a home-based voice assistant, it understands the audio command by detecting the underlying intent and slots, i.e., the defined trigger. Then the parsed trigger allows the system to find the corresponding action.

Bentley *et al.* [17] conducted detailed research on 65 thousand utterances collected from 88 households through their long term usage of Google Home, and here we summarize two insights.

1) *The length of the voice command is limited.* The maximum length of the collected utterances is less than ten words, and the median is four words.
2) *The usage of the voice assistant is highly spatial-temporality related.* Spatially, the voice command is covered by approximately three domains, although different householders have different domains of interest. Furthermore, a typical power-law distribution is observed on command domains (Figure 3). Temporally, the active usage of the voice assistant is 7 AM to 11 PM, and peaks around 5-6 PM, following the human circadian rhythm.

[Motivation 2] The voice commands are driven by the intent, short in the length, and limited in the topic. Thus a caching system at the edge can benefit the current workflow by caching the frequently requested $\{trigger,\ action\}$ pairs.

C. Challenges

The above mentioned two motivations inspire us to propose *CHA*, a caching framework at the edge to support the

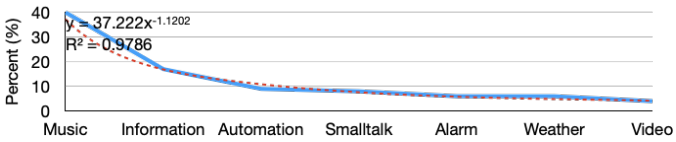


Fig. 3: Home-based voice assistant usage for different domains in a day (red: trendline).

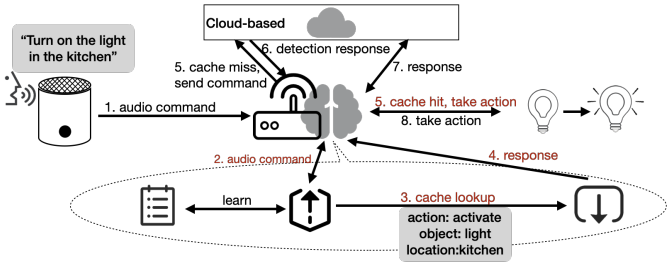


Fig. 4: The request flow in *CHA*.

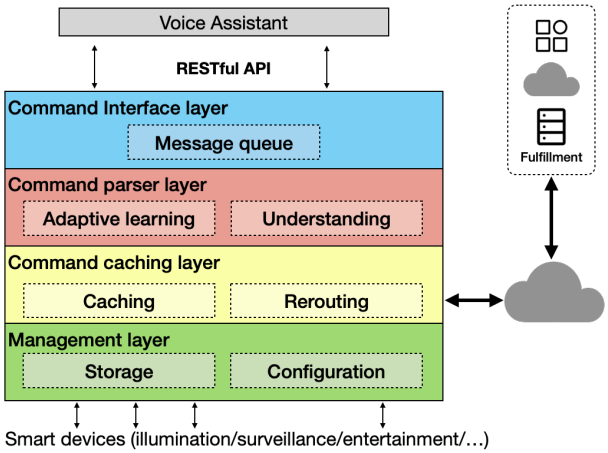


Fig. 5: The *CHA* architecture.

current cloud-based voice assistant systems, and accelerate the response for frequently used intent without intervening with the cloud processing. *CHA*'s workflow is described in Figure 4. It is aimed at offloading cloud processing, reducing the network dependency, and providing fast and stable user experience. We present the key challenges of edge-based voice command processing and describe *CHA*'s solution.

1) Response latency: Compared to web caching systems, extra processing time is introduced to *CHA* because of the understanding part. To diminish its impact on cache miss items, *CHA* needs to parse the voice command quickly. Efficient deep learning methods and models have been proposed to reduce the inference time and squeeze model size for original large models. They have shown acceptable performance on GPU-equipped and powerful CPU-equipped devices [18]–[20]. However, their feasibility on resource-constrained devices needs further discussion.

2) Understanding accuracy: The efficiency of caching depends on the accuracy of voice command understanding. Mistakenly missing a cached item prolongs the response time, and wrongly return a cached item will affect the user experience. An efficient understanding model with high accuracy is the key to support the whole home-based voice assistant system.

3) System efficiency: As a time-sensitive service, *CHA* is expected on the critical path, and the performance must be fast and accurate. Besides, for a smart home environment, the restriction on computation resources is another concern since GPUs or powerful CPUs may not apply in this scenario. Thus the design of *CHA* is supposed to be system efficient and will not bring too much pressure on resource-constrained devices.

To address these challenges, *CHA* leverages a pre-trained language model to achieve state-of-the-art understanding performance. It combines the intent detection and slot filling together in the understanding model to reduce the computational overhead. *CHA* prunes a model to reduce inference processing time and compresses the model and adaptively learns from the feedback and history data to improve the model performance.

III. System Architecture

CHA is designed as a layered architecture with *command interface*, *command parser*, *command caching*, and *management* layers (Figure 5). The command interface layer maintains a message queue to manage the input command in sequence. The command parser layer takes the responsibility to understand commands and improve the prediction performance by learning adaptively. The command caching layer decides the proper routing for the fulfillment of the parsed command, and updates caching accordingly for the up-to-date response. The management layer supports the interaction with connected devices or services according to the user's configuration file.

CHA encapsulates each component as a module. Each module is initialized with the system, and can easily be accessed across the system. The modular design also simplifies the model deployment and replacement in the command parser layer. As the understanding model takes audio as input and output detected action, either the two-step ASR-NLU model or the end-to-end SLU model is applicable.

We first describe the path of voice commands *CHA* before presenting the system design in detail. When the user issues voice commands to the voice assistant, the commands are transmitted through RESTful API to *CHA* and first stored in a message queue in sequence. Then the command parser layer takes the waiting commands from the message queue and parses its meaning. A comprehensible command will be looked at in the caching for valid matching action. In contrast, a command that is incomprehensible or finds no matching action in the caching will be rerouted to the connected cloud service for fulfillment. The corresponding action then goes to control destination devices or services through the management layer for command fulfillment.

In principle, *CHA* is not responsible for generating the fulfillment of itself. It stores executable response from history requests in the caching, which will be queried for each comprehensible command. *CHA* takes two strategies to improve the cache hit rate. One is positive rectification by synthetic

candidate commands; one is passive correction by feedback received from users and the cloud. Both strategies are aimed at improving the understanding model performance.

A. Command Interface Layer

CHA provides an interface to accept the voice commands from the voice assistant. In the implementation of *CHA*, RESTful API is used for audio data and response transmission between the framework and the voice assistant. The voice assistant posts an audio command to *CHA* in the local network, and wait until getting the processing response. When *CHA* receives a command, it first pushes it to the message queue, and evoke the parser layer to process in sequence.

B. Command Parser Layer

The command parser layer is the intelligent part in *CHA*. It integrates an understanding module and an adaptive learning module to provide the understanding capability for voice commands. The understanding module is integrated with lightweight models to target on resource-constraint edge devices. The learning module leverages feedback, and cached history queries to improve the understanding performance.

1) Understanding Model:

Inspired by the mainstream cloud-based voice assistant solution, we design *CHA* with the two-step understanding model, i.e., the ASR-NLU model.

The ASR technology has been well studied in decades, and there are some out-of-shelf solutions available, both online and offline. *CHA* adopts PocketSphinx [21] as the ASR model. Targeting on the deployment on the embedded devices, PocketSphinx is designed with lightweight computation and storage cost. To integrate PocketSphinx in *CHA* with stable and accurate performance, we generate a knowledge-based lexical and language model containing normal vocabularies used in a smart home environment.

The natural language recognized by the ASR model is sent to the NLU model for understanding. As introduced in Section II, the goal of command understanding is to detect the intent as well as fill the slots correctly. They both are considered as classification tasks. Intent detection is the classification on the sentence level, and slot filling is on the word level.

CHA combines the two classification tasks in one model to efficiently use the computation and storage resource at the edge device. This idea has been proved can achieve state-of-the-art performance recently [22], [23]. Besides, to achieve a good understanding performance with limited training data, *CHA* leverages the pre-trained language model BERT [24], as BERT and its variants have shown outstanding performance across different natural language processing (NLP) tasks [25].

Based on the Transformers [26], BERT is constructed by four components: input embedding, multi-head attention-mechanism, feed-forward layers, and output embedding. Using its pre-trained model, and connect the last layer to the classification layer, we can easily fine-tune it to get an NLU model (Figure 6). Leveraging BERT's nature, while filling slots on the word level, the first special token *CLS* can be used to detect the intent of the whole utterance.

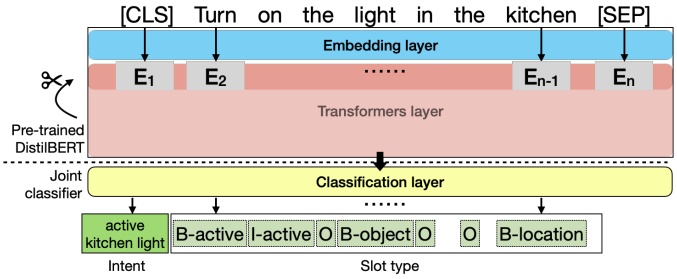


Fig. 6: *CHA*'s NLU model inherits from BERT[1].

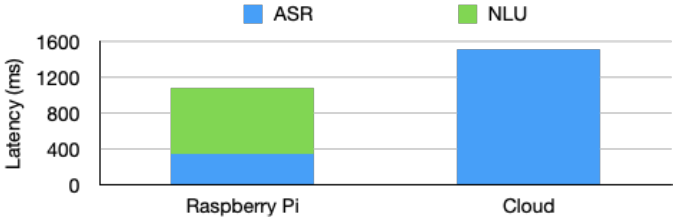


Fig. 7: The latency breakdown before optimization.

Directly deploy BERT, however, is too costly for an edge device. Given there have been some studies on the compression on BERT [27], [28], we build the NLU model in *CHA* based on a pre-trained DistilBERT [29]. DistilBERT takes knowledge distillation [30] to compress the BERT model by 40% with subtle performance degradation, for a light and fast performance.

Targeting on the lightweight understanding model for edge device, we test the response time of the ASR and NLU models mentioned above on a Raspberry Pi. The reason we choose Raspberry Pi is, compared to other costly hardware, Raspberry Pi is affordable and suitable to be considered as the gateway solution for a home environment. Figure 7 compares the latency of processing voice commands on Raspberry Pi and the cloud. The cloud processing time is measured as the round trip latency. The performance on Raspberry pi is severely lagged, and the DistilBERT-based NLU model consumes more than 70% processing time.

As a caching system, the understanding module needs to be efficient enough to keep the processing time low, to diminish the effect on the cache missed item. To achieve acceptable performance when deploying *CHA* on Raspberry Pi and similar resource-constrained devices, here we discuss the model compression strategies considered in *CHA* to build a lightweight understanding module with short processing time.

a) Model Compression:

The DistilBERT-based NLU model has 66 million parameters with 265 MB in model size, making it costly to store and run on a Raspberry Pi. Thus, the model compression in

[1] *CLS* and *SEP* are two inherent tokens in BERT. CLS means classification, SEP means separate preserving for next sentence prediction task.

CHA focuses on reducing the model size and accelerating the inference time.

A series of compression methods have been proposed to facilitate the deployment of deep learning models on edge devices with faster inference speed without the loss of accuracy [31]. Motivated by the design choice that achieving better performance with less required labor, we adopt model pruning to optimize the NLU model on *CHA*.

Pruning reduces the model dimension while keeping the model structure. Jawahar *et al.* [32] found that BERT's lower layers learn basic features, and higher layers learn downstream task-related features. Besides, while the multi-head is proposed to parallel attention mechanism and accelerate the training, Michel *et al.* [33] shows in their work that, during the inference phase keeping one attention head in one layer still preserves the BERT's performance. Given the DistilBERT has 12 attention heads, with 64 dimensions in one head, in total has 768 dimensions.

Inspired by these studies, we prune the DistilBERT by cutting on the number of layers and attention heads, then directly connecting the lower layer to our classification layer before fine-tuning the model, aiming to an efficient while accurate smaller NLU model.

We will further discuss the efficiency of *CHA*'s model compression strategy in Section V.

2) Adaptive Learning:

In addition to the inference time, accuracy is also the critical factor in the design of *CHA* as it can affect the user experience and the caching efficiency. The adaptive learning helps *CHA* to adapt to the change in a smart home by learning from the configuration file and feedback.

The adaptive learning module can be invoked in two ways. One is *active learning*, which gets learning knowledge from the system. Another is *passive learning* that getting feedback from the user and the cloud. The training data are generated jointly from these two ways and fed into the adaptive learning module for improvement.

a) Active Learning: Active learning positively generates training data from the system. This function is reserved for smart home autonomous control. When a new {*trigger, action*} pair is registered to the configuration file, *CHA* searches in the system for registered devices with the same attribute to synthesize command based on the cached history of these devices. For example, when a smart light is installed in the washroom, it's corresponding {*trigger, action*} is registered to the configuration file. The synthetic user commands can be generated based on history commands that used to control the light in the kitchen by replacing the location-related information and labeling with registered configuration.

b) Passive Learning: In *CHA*, low false positive (FP) and low false negative (FN) are required to ensure the user experience. FP comes from user feedback. It happens when a hit item labeled incorrectly by the user. FN comes from cloud feedback. It happens when the system receives the response of a missed item and trying to write it to the cache while identifying the item has already been cached. *CHA* counts the FP and FN to calculate a tolerance score, and trigger passive learning when the score is lower than a threshold. Passive learning collects feedback from the cloud and user to generate training data and new labels.

c) Batch Process: Comparing to the inference process, training a model takes considerable resources. To utilize the resource at the edge, *CHA* adopts batch adaptive learning when the system is in idle state. An idle state of the system can be derived from long term tracking of the usage pattern. By logging the number of commands per hour, *CHA* can construct the distribution of the user commands with Markov Chain Monte Carlo method [34], and choose the predicted idle time to schedule adaptive learning. *CHA* first collects the feedback and configuration data to generate the training data, and save them in the filesystem. The adaptive learning module then works as scheduled to fine-tune the model with the stored aggregated data. Consider that the daily usage is in a limited amount; it is acceptable for the system to do a daily or even weekly update.

C. Command Caching Layer

The command caching layer uses a caching module to provide responses to fulfill cached locally controllable commands and provide a routing module to determine whether the response goes directly to the management layer to control the device or go to the cloud to get the updated response.

1) Caching:

Caching module in *CHA* is designed with the following two purposes. Firstly, as observed in Figure 3 that the real world voice assistant usage follows the power-law distribution, caching at the edge benefits these frequently requested commands. By maintaining the command cache, *CHA* serves these commands without interacting with the cloud. It substantially reduces the service latency and variability by eliminating the additional cost of network transmission. Secondly, command caching serves an essential role in adaptive learning (Section III-B). To correctly understand commands, adaptive learning in *CHA* jointly consider feedback and history commands. Feedback from the cloud is reported by locating a cached item that mistakenly conceived missed. Synthetized data also come from the cached history commands. *CHA* caches {*trigger, action*} of the history command sent from the cloud in one hash table with *(key: trigger, value: action)* format. A parsed command will lookup this hash table for valid action. To make history command available for the learning purpose, the path to recorded audio files is also cached.

When a trigger is modified or removed from the configuration file, *CHA* retrieves its associated values and delete them from the caching. *CHA* employs LRU eviction policy. With an adequately sized cache, *CHA* can utilize the hardware resource efficiently, and avoid the eviction of hot commands.

2) Rerouting:

As an edge-supported caching framework, *CHA* does not provide an entirely offline process. The routing module decides the route of the parsed commands when the corresponding response it not locally available.

trigger: “active kitchen light”
entity: light.kitchen
status: (state == off)
action: state.on

TABLE II: An example in configuration file.

Based on this design, when *CHA* is deployed in the first time, commands in the message queue are sent to the routing part for cloud processing directly. *CHA* caches the parsed result {*trigger, action*} from cloud response for each transaction. After several interactions between the system and the user, *CHA* invokes the understanding module to work.

The policy taken in the routing module is intuitive. A valid cache hit command will be sent to the management layer to control the destination device. The command that missed or expired in cache, or failed to be parsed in the understanding module, will invoke the cloud service from the provided client API. The rerouting part also monitors the response sent back from the cloud, and written it back to the caching to update, and to the management layer for fulfillment.

D. Management Layer

The management layer provides the system configuration file and stores history audio commands in the file system.

The configuration file allows *CHA* to run properly by defining the registered cloud service and local controllable devices. Table II is an example showing a registered {*trigger, action*} pair for kitchen light in the configuration file. The execution of the trigger also checks the current status.

The storage part records history command in audio files, and write the audio path to the cache. Based on the corresponding trigger value, the audio paths are recorded in different hash tables. To avoid the accumulation of audio files wastes the storage space on the edge device, each audio file has an expiration key when writing to the cache. When the key expired, the file will be removed from the system.

IV. Implementation

In this section, we introduce the experiment setup of *CHA*, including the hardware, software, dataset, and the workload.

A. Experimental Setup

The whole system is composed of the device (voice assistant), the edge (where *CHA* deployed on), and cloud (backend support for the voice assistant), as shown in Figure 8. In the experiment implementation, we connect a mic array to a Raspberry Pi as the voice assistant taking audio input. Since resource configurations are different among available edge devices, we choose edge devices that differ on the CPU, CUDA, and memory. Raspberry Pi is an ARM CPU-based, affordable computation resource. Intel Fog Reference Design (FRD) provides powerful CPU configuration, and Jetson AGX Xavier is a GPU-based hardware platform. The detailed parameters are listed in Table III. The cloud-based understanding service for the system is maintained in Google Dialogflow. Google Dialogflow is a cloud-based NLU service created by

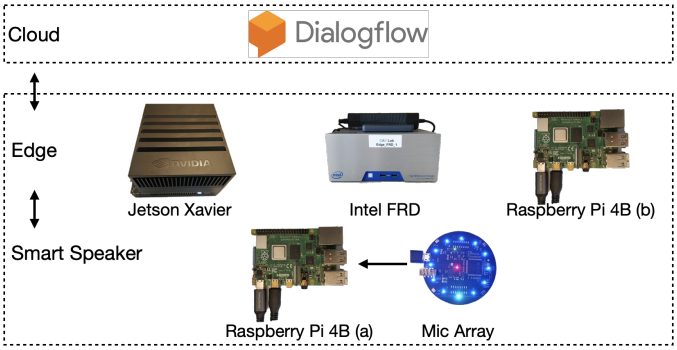


Fig. 8: *CHA* system implementation.

Google to accompany their Speech-to-Text service and provide intelligence to voice assistant systems. It can be queried with either audio or text requests. For the sake of consistency, we train the NLU model in the Dialogflow with the same dataset as *CHA*'s NLU model.

We implement *CHA* in Python 3. We use PyTorch 1.4.0 as the deep learning library to support the understanding module. We use Flask [35] to build RESTful API enabling the data transmission between voice assistant device and *CHA*. We use Redis [36] as the message queue and the cache. The message queue is maintained by a Redis list, and the cache is maintained by several Redis hash tables. As a memory cache, Redis provides low read and write latency for the system. The caching size is set as 2 MB.

B. Dataset

In the experiment, we use Fluent Speech Commands [37], which contains spoken English commands that people interact with a smart home or voice assistant. It covers typical smart home commands, including home automation on various devices at multiple locations, and task management. The data are labeled on intent and slots. Each spoken command is composed of 1 to 9 words, which is consistent with the usage pattern discussed in Section II. There are three slots labeled in each utterance: action, object, and location. The slot types' design follows the same pattern as the trigger configuration we observed in some open-source smart home systems [3]. There are 31 unique intents across the dataset, denoting the specified action expected to be taken by the system. Each intent is expressed in different ways based on human knowledge, ranging from 4 to 24 types of expressions (Table IV). The total number of unique utterances in total is 248. This design is consistent with the observation from [17] that even for a single user, the voice commands are changed from time to time when asking for the same intent. We then train our model based on this dataset and build the workload for cache evaluation from it.

C. Workload Generation

As the voice assistant usage follows the power-law distribution (Figure 3), we simulate the user query in Pareto

Hardware	CPU	GPU	CPU Frequency	Cores	Memory	OS	Cost (USD)
Raspberry Pi mode 4B	ARMv7	N/A	1.5GHz	4	4GB	Linux 4.19.118-v7l+	55.0
Intel Fog Reference Design	Intel Xeon E3-1275	N/A	3.6GHz	4	32GB	Linux 4.15.0-34-generic	N/A
Jetson AGX Xavier	ARMv8	512-core Volta	2.3GHz	8	32GB	Linux 4.9.140-tegra	699.0

TABLE III: Hardware configuration.

Intent (trigger)	Commands
Increase volume	Louder please.
	Turn sound up.
	I can't hear that.
	I need to hear this, increase the volume.
Active kitchen light	Turn on the kitchen light.
	Switch on the kitchen light.
	Kitchen light on.

TABLE IV: Same intent expressed differently.

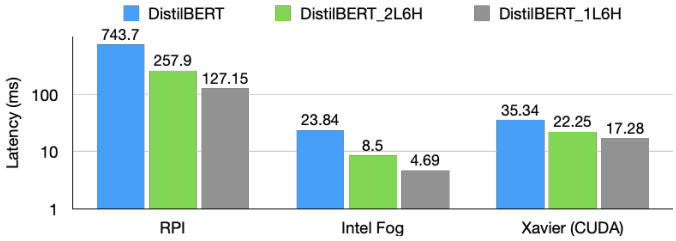


Fig. 9: Pruning benefits inference time.

	Layers	Attention Heads	Model size (MB)	Parameter size (million)
DistilBERT	6	12	265.60	66
DistilBERT_2L6H	2	6	152.20	38
DistilBERT_1L6H	1	6	123.83	30

TABLE V: Pruning benefits model size.

distribution, which belongs to the power-law probability distribution, and has been widely observed in natural phenomena and human activities [38]. The probability distribution formula for Pareto is $f(x, \alpha) = \frac{\alpha}{x^{\alpha+1}}$, where x represents the intent (trigger) of the command, and α is a shape parameter. A higher α leads to a more skewed distribution, where data presents a higher locality. As the real-world usage is close to the Pareto distribution when α is 0.5, we generate the workload to simulate the required intent when α is 0.25, 0.5, and 1.0. In these three cases, the probability of the most frequently used intent in the workload is 62%, 44%, and 34%, respectively. Besides, a uniformly distributed workload is generated and evaluated as well. Each workload has 100 commands, and send the audio request to *CHA* in a random time interval. Given that each intent contains commands expressed in different ways (Table IV), the workload randomly selects one command when an intent is required. At the same time, we take different numbers of transactions to preload as the warm-start to observe its effect on *CHA*.

Besides, we do not restrict the network traffic in the evaluation given the small audio command size (less than 200 KB). The experiment is conducted in a stable network with 10 Mbps download speed and 2 Mbps upload speed, which is a typical Internet speed for at home [39].

V. System Evaluation

In this section, we begin with an analysis of the benefit of the model pruning strategy. Based on the pruned model, we evaluate the performance of *CHA*. First, we compare a home-based voice assistant system with and without *CHA*. Second, we evaluate the resource consumption of *CHA*. Finally, we describe the performance of *CHA* on three edge devices with different hardware configurations. Unless otherwise noted, all the evaluations are conducted on the Raspberry Pi.

A. Model Pruning Strategy

In Section III-B, we propose to compress the NLU model by pruning the number of layers and attention heads to benefit *CHA*'s deployment on resource-restricted edge devices.We find the model and parameter size are linearly related to the number of layers. The same relationship exits between the inference time and the number of layers. Figure 9 and Table V present a few pruned models for demonstration purposes. DistilBERT_2L6H model is cut from 6 to 2 layers (L), DistilBERT_1L6H model is cut from 6 to 1 layer. They both keep half the number of the original attention heads (H). DistilBERT_1L6H reduces the model size by 53%, with 5.8X acceleration in inference.

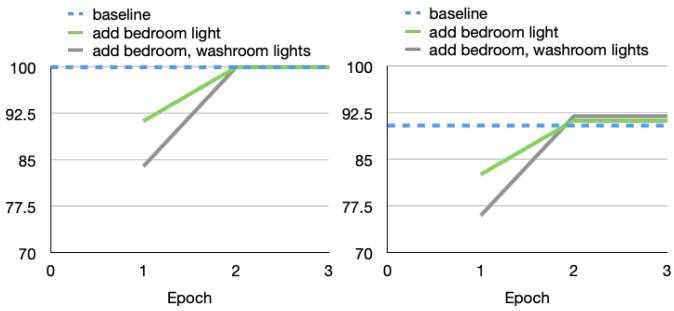
The improvement in model size and inference time comes with the preservation of performance accuracy. On the Fluent Speech Command dataset, the model shows no performance degradation after pruning. Since the size of our dataset is small, we further evaluate the pruning methods on two widely used NLU datasets that containing much larger utterances for the generality purpose. The same pre-trained model and pruning strategy are used on all three datasets. The details about the evaluated datasets and the inference accuracy are listed in Table VI lists the inference accuracy on three datasets.

The listed performance is evaluated on DistilBERT_1L6H model, and the data marked in red shows the performance degradation, i.e., the maximum degradation from pruning. In general, the DistilBERT_1L6H preserves over 98% in intent detection accuracy, and 94% in slot filling comparing to the original model.

Beyond inference, *CHA* runs adaptive learning to com-

	Fluent Speech Command	ATIS [40]	SNIPS [41]
Utterances	248	5871	14484
Intent labels	31	21	7
Slot labels	3	120	72
Intent accuracy	92.0% (0.0%)	96.42% (0.9%)	97.86% (1.4%)
Slot F1 score	96.3% (0.0%)	93.58% (2.0%)	90.04% (5.8%)

TABLE VI: Pruning maintains most of the performance.



(a) Intent Accuracy (b) Sentence Accuracy

Fig. 10: Adaptive learning: adding new device.

pensate for accuracy. Thus we evaluate the pruned model's performance on adaptive learning. When lights in different locations are added to the system (Figure 10), it takes two only epochs for DistilBERT_1L6H to understand the command by training with synthesized data correctly. Beyond that, because synthetic command enriches the diversity of command intents, the performance of sentence accuracy prediction increases.

These experiments show that, with proper model compression such as pruning, a powerful deep learning model can be deployed on resource constraint edge devices and achieve similar high performance with short inference time and small storage size. In addition, the adaptive learning benefits the hit ratio as the model can recognize more commands.

B. System Efficiency

By pruning the BERT-like NLU model in the number of layers, and attention heads, we have proved that the compressed model achieves at least 94% performance as the original model, with the processing time accelerated by almost six times than the original one. The NLU model in *CHA*'s understanding module is based on DistilBERT_1L6H, since it brings most inference acceleration with no degradation on performance in our dataset.

As a caching framework, we first evaluate the caching efficiency of *CHA* from the system latency, cache hit ratio, recall, and precision. Recall and precision are related to the accuracy of the understanding model, higher recall ratio means less FN, higher precision means less FP.

We compare the latency of the home-based voice assistant system working with and without *CHA* in Figure 11(a) when α of the workload is 0.5. Without *CHA*, the voice command is sent directly to the cloud for processing. The cloud response time varies. For demonstration purposes, commands with response time less than 3 seconds are presented.

CHA provides faster and stable experience for users than the cloud. 80% commands are finished by *CHA* in less than 0.8 seconds. In the 80th-percentile finished command, the latency is reduced by 52% to 73%.

The effect of different distributions in user intent and number of preload transactions is shown in Figure 11(b) to 11(d). With more preloaded transactions, the influence of usage intent distribution to the system is getting smaller. The usage pattern

	Uniform	**α = 0.25**	**α = 0.5**	**α = 1.0**
Preload = 5	74%	82%	84%	88%
Preload = 10	77%	86%	88%	88%
Preload = 20	85%	85%	87%	91%

TABLE VII: Hit ratio of *CHA*.

	Uniform	**α = 0.25**	**α = 0.5**	**α = 1.0**
Preload = 5	93%	96%	99%	100%
Preload = 10	93%	100%	95%	100%
Preload = 20	95%	93%	99%	99%

TABLE VIII: Precision of *CHA*.

that has a higher diversity in intents benefits more from a small number of preloaded transactions. For the high locality usage pattern, *CHA* can start the understanding module after a few transactions have been completed in the cloud.

Table VII lists the hit ratio and precision in different cases. Higher locality intent distribution has a higher hit ratio of voice commands. Preloading also helps the system to have a higher hit ratio. Recall and precision are related to the accuracy of the understanding model, higher recall ratio means less FN, higher precision means less FP. Here, we assume the response from the cloud is the ground truth to calculate precision and recall value. Across our evaluation, the recall is 1.0 since no FN happens. The precision is higher than 92% (Table VIII). For the FP item, *CHA* rectifies them by the feedback from the user.

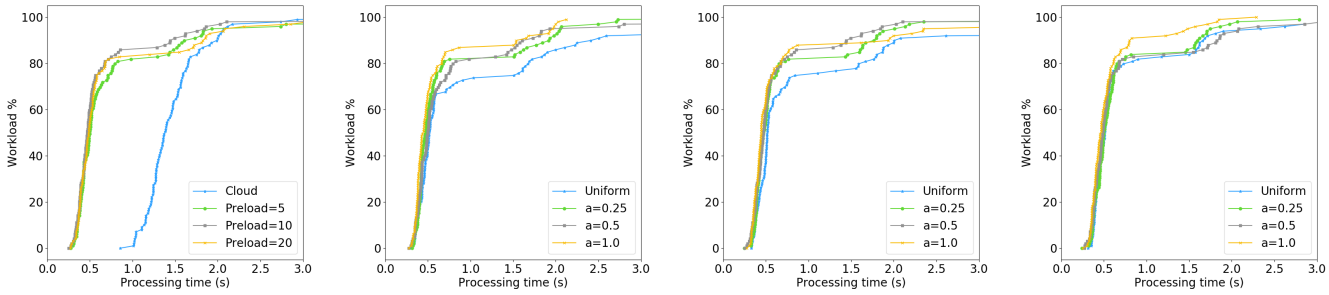
These experiments show that the deployment of *CHA* on Raspberry Pi can provide a fast and stable response for the home-based voice assistant systems by the lightweight understanding module and a caching system.

C. Resource Consumption

Beyond the response time and understand accuracy, the resource consumption consumed on the resource constraint edge device is one important factor in evaluating *CHA*'s availability. We consider the resource consumption by analyzing the memory, CPU usage. We use *top* to get the percentage of memory and CPU utilization, *memory-profile* to get memory size used in a finer time granularity. We compare the resource consumption of *CHA* on Raspberry Pi with a baseline system that using unpruned DistilBERT as the NLU model. The resource consumption of a Flask implemented gateway is also measured as a comparison.

The lightweight understanding module considerably lowers the memory and CPU usage comparing to the baseline. It consumes three times more memory and CPU than a Gateway API. As an intelligent system, its resource consumption is acceptable for the edge device. In the system loading phase, *CHA* can load each module and be ready to receive commands 2X faster than the baseline. The maximum memory *CHA* consumes is lower than the baseline's system running period.

These experiments show that the execution of *CHA* does not cause new bottlenecks in the resource utilization on resource-constraint edge devices.



(a) $\alpha = 0.5$ (b) Preload commands = 5 (c) Preload commands = 10 (d) Preload commands = 20

Fig. 11: Latency with different workload and preload strategies.

	Memory utilization (%)	CPU utilization (%)
Baseline	11.88	23.41
Gateway	2.74	3.80
CHA	**8.03**	**11.78**

TABLE IX: The resource consumption on Raspberry Pi.

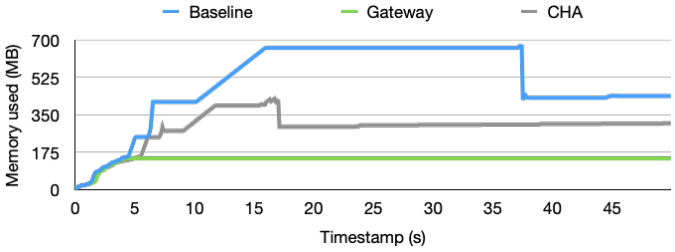
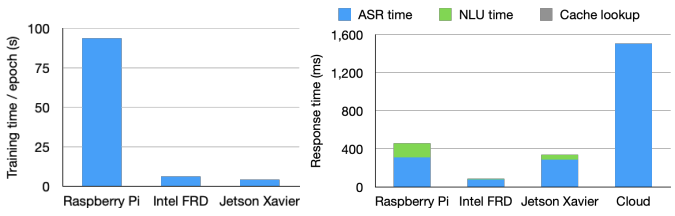


Fig. 12: Memory footprint in *CHA*'s loading phase.

D. *CHA on Different Edge Devices*

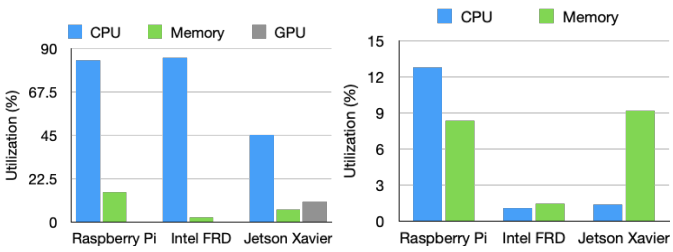
Finally, we deploy *CHA* on three edge devices configured differently in hardware (Table III), to explore the generality of *CHA*. We evaluate the responsiveness of *CHA* on each device, and resource consumption *CHA* required to provide stable performance. For the Jetson Xavier where CUDA is equipped, we analyze its GPU usage as well, and use *tegrastats* to get GPU usage on it.

CHA's performance on different platforms is evaluated from both command processing and adaptive learning parts. The command processing is the model inference phase, during which the system receives an audio command from voice



(a) Adaptive learning (b) Command processing

Fig. 13: Responsiveness on three edge devices.



(a) Adaptive learning (b) Command processing

Fig. 14: Resource utilization on three edge devices.

assistant to parse. The adaptive learning is the model training phase, during which the system fine-tunes the understanding model with newly generated data.

Figure 13 compares the responsiveness in two phases. Raspberry Pi takes a longer time to process in training and inference due to the computation resource it equipped is less powerful than the other two. It takes around 1.5 minutes to train one epoch in adaptive learning, while Intel FRD takes 6 seconds and Jetson Xavier takes 4 seconds. In Figure 10, we presented the number of epochs required in adaptive learning is quite small; thus, it is acceptable for a resource-limited device to execute the adaptive learning module. Depending on aggregated utterances that need to be rectified, *CHA* can schedule the adaptive learning in a daily or weekly manner. Figure 13(b) demonstrates the end-to-end processing time on edge devices. For cached commands, *CHA* reduces the processing time by 70%, 94%, and 77% than the cloud-based system on Raspberry Pi, Intel FRD, and Jetson Xavier respectively. Thus, for a cache missed item, *CHA* keeps the low response overhead.

In terms of resource utilization (Figure 14), adaptive learning takes a considerable proportion of CPU usage across three devices. Jetson Xavier takes 50% less usage on CPU because the model is moved to CUDA for training. The average resource utilization in command processing is less than 13%, 2%, and 9% on Raspberry Pi, Intel FRD, and Jetson Xavier for both CPU and memory. Since the inference workload is low, GPU utilization on Jetson Xavier is nearly zero after

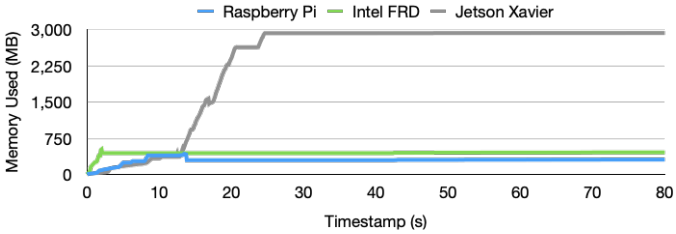


Fig. 15: Memory footprint on three edge devices.

	Layers	Model size (MB)	Parameter size (million)
BERT	12	438.10	110
BERT_1L	1	126.19	30
ALBERT	1	46.87	12

TABLE X: Model size of BERT-like models.

averaging, Figure 14(b) only shows CPU and memory usage.

The memory footprint on three devices since *CHA* starts to load is shown in Figure 15. The system loading phase takes 13 seconds, 2 seconds, and 24 seconds on Raspberry Pi, Intel FRD, and Jetson Xavier. The high memory usage and longer loading time observed in Jetson Xavier results from a fixed overhead to the CUDA runtime, as the NLU model running on CUDA for inference.

In summary, the design of *CHA* makes it applicable to be deployed on edge devices with different hardware configurations. Intel FRD shows outperforming capability in both model training and inference phase, with the shortest processing time and lowest resource utilization. Raspberry Pi, although limited in CPU and memory resource, is also capable of deploying *CHA* and provide stable and fast performance for home-based voice assistant systems.

VI. Discussion

In this section, we discuss the challenge of edge-based voice command processing based on our observations during the deployment and evaluation of *CHA*.

Observation 1: *For BERT and its variants, layer pruning is a simple and effective compression solution that making it possible to deploy on edge devices with subtle performance degradation.*

In Section V, we evaluate the benefits of model pruning on the pre-trained DistilBERT. To explore the generality of the layer pruning on the BERT-like structure, we adopt the same pruning strategy on pre-trained BERT and ALBERT [42] as well. Table X is the model size before and after layer pruning. ALBERT has small model size because it only has one layer, and iterates this layer 12 times in execution. So our layer pruning on ALBERT is reducing the iteration times. The performance of layer-pruned BERT and ALBERT models running on our tested dataset maintained 96% of the original model's performance (Table XI).

Observation 2: *To deploy an efficient deep learning model on devices configured differently in hardware, quantization is not a one-for-all solution. It is affected by available deep learning framework, hardware, quantization engine.*

	BERT_1L	ALBERT_1L
Intent accuracy	92.0% (4.0%)	96.0% (0%)
Slot F1 score	96.3% (0.0%)	96.3% (0.0%)

TABLE XI: Performance of layer-pruned BERT-like models.

	Raspberry Pi	Intel FRD	Jetson Xavier
Inference time	737.0 ms (127.2 ms)	41.4 ms	83.0 ms
Model size	15.9 MB (123.8 MB)		
Parameter size	3 million (30 million)		

TABLE XII: SLU model (red: DistilBERT_1L6H).

Given that in BERT's structure, the feed-forward layer takes a considerable proportion. We attempted post-training integer quantization on linear layers of the fine-tuned DistilBERT to compress the model. With the built-in quantization tool provided by PyTorch 1.4.0, the model size is reduced by 48%, and the inference time reduced by 48.5% on Intel FRD. A similar improvement is observed on other X86-based machines. On a Macbook Pro (2.8 GHz Quad-Core Intel Core i7), quantization reduces the inference time by 89%. However, similar performance was not shown on two ARM-based devices which using QNNPACK [43] as the backend quantization engine. Meanwhile, since the CUDA-based quantization in PyTorch has not been supported, for a CUDA-based device, models need to switch from PyTorch to TensorRT for quantization, making steep learning curve for efficient model deploying at the edge.

Observation 3: *For end-to-end SLU models, efficient voice processing at the edge is challenging due to the dense and informative model structure.*

From Figure 7 and 13(b), we find that after compression, the ASR engine becomes the most significant time-consuming part on different platforms and needs to be optimized. It brings double labors to model compression since ASR and NLU are back-ended with different techniques. One alternative way is adopting the end-to-end SLU model and make optimization on it. The end-to-end SLU model has been proposed recently [37], [44], where the model takes the voice command and directly output the detected slot and intent text. Convenient by *CHA*'s modular design, we effortlessly replace the ASR-NLU model with one SLU model [37] and get its processing performance as shown in Table XII. Unlike DistilBERT, where the model is large in terms of model size and parameter number, the SLU model is quite small and dense. It is significantly smaller than the NLU model (87% smaller in model size, 90% smaller in parameters), while it takes more than seven times to process than the latter on Raspberry pi. We found that *CHA*'s model compression does not apply to the SLU model. Due to the difference in model structure and components. The SLU model processes phoneme and word feature in sequence, and merely pruning part of the weights will either show no effects on the model size or significantly hurts the accuracy. We save it for future work to explore the optimization of SLU models on edge devices.

VII. Related work

A. Caching System

As computation-intensive and latency-sensitive applications, intelligent services leverages cache to accelerate processing time. Cachier [45], [46] is proposed to minimize latency for image recognition services by inserting a specified cache between the user and the cloud. MUVR [47] is a framework designed to serve multiple virtual reality users at the same point of interest by maximizing the efficiency of utilizing the edge cloud's computation and communication resources. Potluck caches duplicated processing results for vision workloads across applications [48]. Zhang *et al.* [49] leverages a deep reinforcement learning algorithm to find the proactive caching policy for multi-view 3D videos. Besides these caching services designed for computer vision tasks, some work focuses on caching the reusable processing results for cognitive services. Elbamby *et al.* [50] proposes to minimize the latency by proactively caching popular computing results. As a prediction services system, Clipper leverages function cache to store the intermediate results to reduce the prediction latency and improve the throughput [51]. FoggyCache caches computation processing for similar contextual data across devices and shows promising results on powerful GPU-based machines [52].

B. Spoken Language Understanding

SLU is the technology that supports voice assistant systems. The conventional solution is composed of ASR and NLU modules [41], [53]. DeepSpeech [54] and Kaldi [55] are two well known ASR models. PocketSphinx [21] is a lightweight ASR model designed for embedded devices. The performance of ASR models rely on language package, consider the storage and deployment cost, current voice recognition systems mainly rely on the cloud resource. The studies in NLP get significant development after BERT [24] has been proposed. BERT provides a pre-trained model that can be used for different downstream tasks and show promising performance on many of its variants [25]. To predict the intent and slots at the same time, Chen *et al.* [23] proposed jointBERT. An alternative solution burgeoning recently is end-to-end SLU [37], [56]–[58]. In this type of model, the speech is represented from phonemes to words to meanings. Because of the high dimensional and high variable character of speech signals, the SLU model's development is challenging. For current voice assistant systems, the two-step ASR-NLU model is the mainstream solution.

C. Edge Intelligence

Bringing intelligence to edge needs efficiency deep learning to improve hardware efficiency on resource constraint devices. Large pre-trained models, such as BERT, are usually huge in model size and expensive in deploy cost [59], different model compression strategies are proposed. Model pruning is one typical solution that pruning the BERT model during pre-trained phase [28], cutting on number of attention heads [33], reducing the layer dimensions [60], extracting sub-network from the complete model [61], etc. Knowledge distillation [27], [29], [62] compresses the model according to the principle that, a well-performed large model contains rich generalization knowledge in softmax output. Thus distillation transfers the generalization knowledge from a well-performed large model to a small model by training the latter with a loss over the output from the large model. Quantization provides a smaller model size by reducing the precision of the model parameters. Quantization methods vary on different model structures [63], [64]. The optimization on *CHA* is motivated by these studies. Besides, the efficient neural network architecture is also a solution for edge intelligence, HAT [65] is a hardware-aware Transformers network for NLP tasks that can generate specialized sub-model to deploy on different hardware based on the constraint analysis. Although efficient deep learning has been well studied in NLP and computer vision field [18]–[20], [66], compression on ASR and SLU models is still a big challenge [67].

VIII. Conclusion

Voice assistant systems are becoming immersive nowadays as a new way for people to interact with the machine. However, current voice assistant systems are purely based on the cloud to process audio commands and fulfillment. This workflow can prevent users from having a stable and fast user experience due to the unpredictable situation in the cloud server and the network. It also brings unnecessary data transmission through the internet when the voice commands are locally controllable.

In this paper, we first analyze cloud-based voice assistant systems and find two drawbacks: long latency and unstable response. To address the two drawbacks, we propose *CHA*, a caching framework to provide fast and stable performance for home-based voice assistant systems. *CHA* is constructed with a layered architecture with a modular design. It provides an understanding module to parse voice commands, a caching module to cache history commands with corresponding fulfillment from the cloud, a routing module to dispatch voice commands and a configuration file to provide locally controllable information. It maintains a message queue to manage received voice commands. It also integrates an adaptive learning module and history storage unit to rectify the parsing accuracy. In order to make *CHA* applicable on low-cost, resource-restricted edge device, we use joint classification and model pruning on a pre-trained language model for *CHA* to accelerate the inference and reduce the resource consumption while maintaining high accuracy. The evaluation of *CHA* on three edge devices that differ in hardware configuration shows that as the support framework deployed at the edge, *CHA* is capable of providing fast, stable, and accurate responses for home-based voice assistant systems with efficient system resource utilization. Moreover, resource-constrained edge devices can process voice commands for voice assistant systems. On Raspberry Pi, *CHA* provides a 70% acceleration in response compared to cloud processing, with resource consumption less than 13% in CPU and 9% in memory on average.

REFERENCES

[1] S. Analytics, "Global smart speaker vendor & os shipment and installed base market share by region: Q4 2019," 2020, https://www.strategyanalytics.com/access-services/devices/connected-home/smart-speakers-and-screens/market-data/report-detail/global-smart-speaker-vendor-os-shipment-and-installed-base-market-share-by-region-q4-2019.

[2] V. Research, "Smart speaker consumer adoption report april 2020," 2020, https://research.voicebot.ai/report-list/smart-speaker-consumer-adoption-report-2020/.

[3] H. Assistant, "Home assistant," 2020, https://www.home-assistant.io/.

[4] H. Assistant, "Ada," 2020, https://github.com/home-assistant/ada.

[5] OpenHAB, "Google assistant action," 2020, https://www.openhab.org/docs/ecosystem/google-assistant/.

[6] Google, "Dialogflow," 2020, https://dialogflow.com/.

[7] Amazon, "Amazon lex," 2020, https://aws.amazon.com/lex/.

[8] Microsoft, "Language understanding (luis)," 2019, https://www.luis.ai/.

[9] Facebook, "Wit.ai," 2020, https://wit.ai/.

[10] IBM, "Ibm watson products and solutions," https://www.ibm.com/watson/services/conversation/.

[11] OpenAI, "AI and compute," 2020, https://openai.com/blog/ai-and-compute/.

[12] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[13] X. Zhang, Y. Wang, S. Lu, L. Liu, W. Shi *et al.*, "OpenEI: An open framework for edge intelligence," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1840–1851.

[14] Amazon, "Amazon digital and device forum," 2020, https://www.amazonforum.com/s/global-search/slow%20response.

[15] Google, "Google nest help," 2020, https://support.google.com/googlenest/search?q=slow+response&from_promoted_search=true.

[16] G. Cloud, "Speech-to-Text," 2020, https://cloud.google.com/speech-to-text.

[17] F. Bentley, C. Luvogt, M. Silverman, R. Wirasinghe, B. White, and D. Lottridge, "Understanding the Long-Term Use of Smart Speaker Assistants," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 2, no. 3, pp. 1–24, Sep. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3264901

[18] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5mb model size." [Online]. Available: http://arxiv.org/abs/1602.07360

[19] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications." [Online]. Available: http://arxiv.org/abs/1704.04861

[20] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. Salt Lake City, UT: IEEE, Jun. 2018, pp. 4510–4520. [Online]. Available: https://ieeexplore.ieee.org/document/8578572/

[21] D. Huggins-Daines, M. Kumar, A. Chan, A. W. Black, M. Ravishankar, and A. I. Rudnicky, "Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices," in *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, vol. 1. IEEE, 2006, pp. I–I.

[22] C.-W. Goo, G. Gao, Y.-K. Hsu, C.-L. Huo, T.-C. Chen, K.-W. Hsu, and Y.-N. Chen, "Slot-gated modeling for joint slot filling and intent prediction," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics, pp. 753–757. [Online]. Available: http://aclweb.org/anthology/N18-2118

[23] Q. Chen, Z. Zhuo, and W. Wang, "BERT for joint intent classification and slot filling." [Online]. Available: http://arxiv.org/abs/1902.10909

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv:1810.04805 [cs]*, May 2019, arXiv: 1810.04805. [Online]. Available: http://arxiv.org/abs/1810.04805

[25] SuperGLUE, "Superglue leaderboard," 2020, https://super.gluebenchmark.com/leaderboard.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, . Kaiser, and I. Polosukhin, "Attention is All you Need," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 5998–6008. [Online]. Available: http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[27] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "TinyBERT: Distilling BERT for natural language understanding." [Online]. Available: http://arxiv.org/abs/1909.10351

[28] M. A. Gordon, K. Duh, and N. Andrews, "Compressing BERT: Studying the effects of weight pruning on transfer learning." [Online]. Available: http://arxiv.org/abs/2002.08307

[29] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter." [Online]. Available: http://arxiv.org/abs/1910.01108

[30] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network." [Online]. Available: http://arxiv.org/abs/1503.02531

[31] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks." [Online]. Available: http://arxiv.org/abs/1710.09282

[32] G. Jawahar, B. Sagot, and D. Seddah, "What does BERT learn about the structure of language?" in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, pp. 3651–3657. [Online]. Available: https://www.aclweb.org/anthology/P19-1356

[33] P. Michel, O. Levy, and G. Neubig, "Are sixteen heads really better than one?" [Online]. Available: http://arxiv.org/abs/1905.10650

[34] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng, *Handbook of markov chain monte carlo*. CRC press, 2011.

[35] Flask, "Flaskrestful," 2020, https://flask-restful.readthedocs.io/en/latest/.

[36] RedisLabs, "redis," 2020, https://redis.io/.

[37] L. Lugosch, M. Ravanelli, P. Ignoto, V. S. Tomar, and Y. Bengio, "Speech model pre-training for end-to-end spoken language understanding." [Online]. Available: http://arxiv.org/abs/1904.03670

[38] L. A. Adamic and B. A. Huberman, "Zipf's law and the internet." *Glottometrics*, vol. 3, no. 1, pp. 143–150, 2002.

[39] F. C. Commission, "Household broadband guide," 2020, https://www.fcc.gov/consumers/guides/household-broadband-guide.

[40] G. Tur, D. Hakkani-Tur, and L. Heck, "What is left to be understood in ATIS?" in *2010 IEEE Spoken Language Technology Workshop*. IEEE, pp. 19–24. [Online]. Available: http://ieeexplore.ieee.org/document/5700816/

[41] A. Coucke, A. Saade, A. Ball, T. Bluche, A. Caulier, D. Leroy, C. Doumouro, T. Gisselbrecht, F. Caltagirone, T. Lavril, M. Primet, and J. Dureau, "Snips voice platform: an embedded spoken language understanding system for private-by-design voice interfaces." [Online]. Available: http://arxiv.org/abs/1805.10190

[42] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A lite BERT for self-supervised learning of language representations." [Online]. Available: http://arxiv.org/abs/1909.11942

[43] PyTorch, "Qnnpack," 2020, https://github.com/pytorch/QNNPACK.

[44] S. Bhosale, I. Sheikh, S. H. Dumpala, and S. K. Kopparapu, "End-to-end spoken language understanding: Bootstrapping in low resource scenarios," in *Interspeech 2019*. ISCA, pp. 1188–1192. [Online]. Available: http://www.isca-speech.org/archive/Interspeech_2019/abstracts/2366.html

[45] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan, "Cachier: Edge-caching for recognition applications," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 276–286.

[46] U. Drolia, K. Guo, and P. Narasimhan, "Precog: prefetching for image recognition applications at the edge," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. San Jose California: ACM, Oct. 2017, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3132211.3134456

[47] Y. Li and W. Gao, "MUVR: Supporting multi-user mobile virtual reality with resource constrained edge cloud," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 1–16.

[48] P. Guo and W. Hu, "Potluck: Cross-Application Approximate Deduplication for Computation-Intensive Mobile Applications," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. Williamsburg VA USA: ACM, Mar. 2018, pp. 271–284. [Online]. Available: https://dl.acm.org/doi/10.1145/3173162.3173185

[49] Z. Zhang, Y. Yang, M. Hua, C. Li, Y. Huang, and L. Yang, "Proactive caching for vehicular multi-view 3d video streaming via deep reinforcement learning," *IEEE Transactions on Wireless Communications*, 2019.
[50] M. S. Elbamby, M. Bennis, and W. Saad, "Proactive edge computing in latency-constrained fog networks," in *2017 European conference on networks and communications (EuCNC)*. IEEE, 2017, pp. 1–6.
[51] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency Online Prediction Serving System," 2017, pp. 613–627. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw
[52] P. Guo, R. Li, B. Hu, and W. Hu, "FoggyCache: Cross-Device Approximate Computation Reuse," *Living on the Edge*, p. 16, 2018.
[53] G. Mesnil, Y. Dauphin, K. Yao, Y. Bengio, L. Deng, D. Hakkani-Tur, X. He, L. Heck, G. Tur, D. Yu, and G. Zweig, "Using Recurrent Neural Networks for Slot Filling in Spoken Language Understanding," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 23, no. 3, pp. 530–539, Mar. 2015, conference Name: IEEE/ACM Transactions on Audio, Speech, and Language Processing.
[54] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng, "Deep speech: Scaling up end-to-end speech recognition." [Online]. Available: http://arxiv.org/abs/1412.5567
[55] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely, "The Kaldi Speech Recognition Toolkit," 2011, conference Name: IEEE 2011 Workshop on Automatic Speech Recognition and Understanding, Number: CONF, Publisher: IEEE Signal Processing Society. [Online]. Available: https://infoscience.epfl.ch/record/192584
[56] D. Serdyuk, Y. Wang, C. Fuegen, A. Kumar, B. Liu, and Y. Bengio, "Towards end-to-end spoken language understanding." [Online]. Available: http://arxiv.org/abs/1802.08395
[57] Y.-P. Chen, R. Price, and S. Bangalore, "Spoken language understanding without speech recognition," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, pp. 6189–6193. [Online]. Available: https://ieeexplore.ieee.org/document/8461718/
[58] V. Renkens and H. Van hamme, "Capsule networks for low resource spoken language understanding." [Online]. Available: http://arxiv.org/abs/1805.02922
[59] O. Sharir, B. Peleg, and Y. Shoham, "The cost of training NLP models: A concise overview." [Online]. Available: http://arxiv.org/abs/2004.08900
[60] J. S. McCarley, R. Chakravarti, and A. Sil, "Structured pruning of a BERT-based question answering model." [Online]. Available: http://arxiv.org/abs/1910.06360
[61] A. Fan, E. Grave, and A. Joulin, "Reducing transformer depth on demand with structured dropout." [Online]. Available: http://arxiv.org/abs/1909.11556
[62] C. Xu, W. Zhou, T. Ge, F. Wei, and M. Zhou, "BERT-of-theseus: Compressing BERT by progressive module replacing." [Online]. Available: http://arxiv.org/abs/2002.02925
[63] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8bert: Quantized 8bit BERT." [Online]. Available: http://arxiv.org/abs/1910.06188
[64] S. Shen, Z. Dong, J. Ye, L. Ma, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "Q-BERT: Hessian based ultra low precision quantization of BERT." [Online]. Available: http://arxiv.org/abs/1909.05840
[65] H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han, "HAT: Hardware-aware transformers for efficient natural language processing." [Online]. Available: http://arxiv.org/abs/2005.14187
[66] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment." [Online]. Available: http://arxiv.org/abs/1908.09791
[67] . Dudziak, M. S. Abdelfattah, R. Vipperla, S. Laskaridis, and N. D. Lane, "ShrinkML: End-to-end ASR model compression using reinforcement learning." [Online]. Available: http://arxiv.org/abs/1907.03540