Pipeline Decomposition of Speech Decoders and Their Implementation Based on Delayed Evaluation

Takahiro Shinozaki*, Sadaoki Furui†, Yasuo Horiuchi* and Shingo Kuroiwa*
* Chiba University, Chiba, Japan www.ailab.tj.chiba-u.jp
† Tokyo Institute of Technology, Tokyo, Japan http://www.agl.titech.ac.jp/

***Abstract*—For large vocabulary continuous speech recognition, speech decoders treat time sequence with context information using large probabilistic models. The software of such speech decoders tend to be large and complex since it has to handle both relationships of its component functions and timing of computation at the same time. In the traditional signal processing area such as measurement and system control, block diagram based implementations are common where systems are designed by connecting blocks of components. The connections describe flow of signals and this framework greatly helps to understand and design complex systems. In this research, we show that speech decoders can be effectively decomposed to diagrams or pipelines. Once they are decomposed to pipelines, they can be easily implemented in a highly abstracted manner using a pure functional programming language with delayed evaluation. Based on this perspective, we have re-designed our pure-functional decoder Husky proposing a new design paradigm for speech recognition systems. In the evaluation experiments, it is shown that it efficiently works for a large vocabulary continuous speech recognition task.**

I. Introduction

Automatic speech decoders are software that takes a speech waveform and outputs a sequence of recognized words. Recent decoders for large vocabulary continuous speech recognition are based on statistical models such as N-grams and hidden Markov models (HMM) that have more than millions of parameters. The codes that implement the decoders are written in procedural programming languages such as C and C++ [1], [2], [3]. Because these programs have to handle component functions that treat the models considering context information and timing of computations at the same time, their codes tend to be large and complex and it is very difficult to understand. This is particularly a problem for researches who try to implement a new idea modifying an existing code and for students who want to learn speech recognition. To promote research of the area and to support future developments for more flexible and intelligent systems, a new design framework is expected that allows easy understanding and modification of complex systems.

Speech decoders can be regarded as a kind of very complex non-linear digital signal processing systems since the input and the output are time series. For the traditional signal processing such as measurement and system control, block diagram based implementations are popular [4]. In there, systems are designed by connecting blocks of components such as band-pass filter and fast Fourier transform. The data flow and the functions of blocks are defined separately, and the timing of computation for sequential data is taken care of by the system. This helps a lot to understand and design complex systems. However, such diagram based design is so far limited to the traditional signal processing since it lacks flexibility to describe more complex general systems [5].

A closely related programming paradigm is the pure functional programming with delayed or lazy evaluation, which is known to have high abstraction and modularization ability [6]. Delayed evaluation gives a means to separate connections of functions and actual timing of the computation. This makes it possible to define a large or even infinite data structure and pass it to other functions and still achieve efficient execution. By applying this mechanism to list structured data, pipeline of data processing is efficiently described [7]. Different from the diagram approach, pure functional programming languages are of general purpose and are flexible to describe arbitrary systems.

From a programmer point of view, the pipeline based system design gives a good perspective to write efficient codes using pure functional programming languages or otherwise it is often very difficult. Actually, we have previously developed a speech decoder Husky using a pure functional language Haskell [8]. While we have successfully demonstrated that very abstracted and compact description is possible by utilizing the pure functional programming framework, we had a difficulty to tune the computational performance. Because of that, the decoder required huge amount of memory and CPU time.

In summary, the existing program codes of speech decoders are complex because they have to handle both relationships of its component functions and timing of computation at the same time. If a system is decomposed to pipelines, then it will help to understand the system. Moreover, it can be easily and efficiently implemented by using a pure functional programming language with delayed evaluation. Therefore, if speech decoders are decomposed to pipelines, it is expected to be very useful. In this research, we show how speech decoders are decomposed to pipelines. In the experiment, a new version of the Husky decoder is implemented based on a pipeline schematic design and the performance is evaluated.

The organization of the paper is as follows. In Section II, the pure functional programming is briefly reviewed. In Sec-

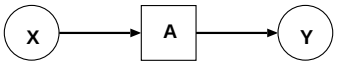

Fig. 1. Transformation of a sequential data X to Y by a function A that applies *afn* to each of the elements.

tion III, pipeline decomposition of speech recognition systems are explained. Experimental conditions are described in Section IV and the results are shown in Section V. Finally, summary and conclusions are given in Section VI.

II. Pure functional programming

The history of programming language had been started with machine code, and the today's standard is procedural programming. In procedural programming, a program has a state represented by variables, which are updated step by step in the program execution. Depending on the state, the same function with the same arguments can result in different values at different times.

Pure functional programming originated from the lambda calculus. In contrast to procedural programming, there is no variable in pure functional programming and the functions have no side effect. Functions can be arguments as well as returned values of other functions. Examples of languages for pure functional programming are Haskell [9] and Clean [10].

One of the characteristics of pure functional programming is that it allows effective use of delayed evaluation. In procedural programming, arguments of a function is evaluated to values before the function is called with them. This strategy is called strict evaluation. Conversely, when evaluation of arguments is delayed until the last possible moment, it is called delayed evaluation. With delayed evaluation, it is possible to define a large or even infinite data structure and pass it to other functions and still achieve efficient execution. The match of delayed evaluation with variable substitution is unfortunately not good, and it is specifically used in pure functional programming.

When delayed evaluation is applied to a list that represents sequential data, it can be regarded as a pipeline separating the relationship between functions and the timing of computation. For example, let X be a sequence of T values $x_1, x_2, \cdots, x_T$. Let Y be a sequence of values that are obtained by applying a transformation A to X, where A applies an affine transformation *afn* to each component of the given sequence as shown in Figure 1. For procedural programming, "*for*" loop would be used to describe the procedure. The procedure would take the list X and applies the transformation *afn* element by element, and then returns a list Y that contains the results. For functional programming, there is no syntax to represent a loop since it does not have variable. Instead, "*map*" function would be used that takes a function as the first argument and a list as the second. It applies the function to each of the list elements. A pseudo Haskell code is as follows.

```
listY = map afn listX
```

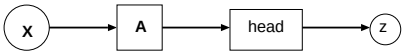

Fig. 2. Composition of *afn* and *head*.

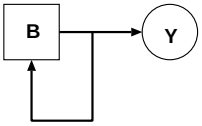

Fig. 3. Example of a pipeline having a self-loop.

In the code, *listX* denotes X and *listY* denotes Y. Suppose at this moment it is not known how the result is used.

Now, assume that "*head*" is a function that takes a list and returns the first element. When the first element of Y is wanted, it can be described as a composition of the functions A and *head* as shown in Figure 2. Although, such a straightforward composition is not efficient in the procedural programming since *afn* is first applied to all the list elements and then all but the first one are discarded. In fact, it is not possible for procedural programming to eliminate the redundant computation keeping *afn* and *head* as independent components without adding extra codes. For functional programming, this is straightforward as shown in the following pseudo code.

```
funcA = map afn
z = head (funcA listX)
```

In the code, *funcA* represents a composed function *(map · afn)* that takes and returns a list. Thanks to delayed evaluation, *afn* is executed only for the first element and no extra computation is performed for other elements. It works efficiently even when the input list is infinite. This is one of the advantages of pure functional programming that contributes to abstraction and modularization of codes.

Pipelines can have loops. Figure 3 is the simplest example, where B is a function that takes and returns a list. The corresponding pseudo code is as follows.

```
listY = funcB listY
```

If B is a function that returns a list in which 1 is appended at the head of the given list, then the output Y is a list of infinite sequence of 1.

Figure 4 is a more general example of a pipeline. In the figure, B is a function that takes two lists and returns a list, and C is a function that takes and returns a list. It is described by the following pseudo code.

```
listY = funcC (funcB listX listY)
```

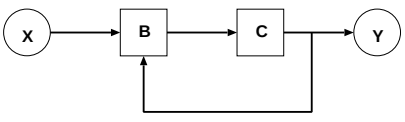

Fig. 4. Example of a pipeline having two functional blocks and a loop.

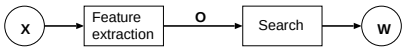

Fig. 5. Top level pipeline representation of a basic speech decoder.

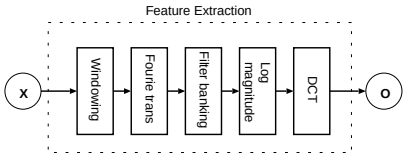

Fig. 6. Pipeline for feature extraction.

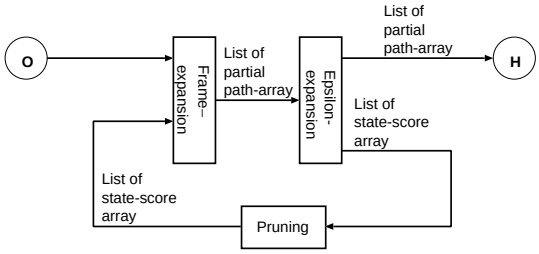

Fig. 7. Pipeline for Viterbi beam search.

When pipelines are used, flow of data must be considered. For example, if a component function takes a list and returns the reversed one, it requires memory that is linear to the length of the input list, and it does not work for infinite input. This is the requirement by the nature of the computation of sequential data itself, not a limitation of the languages.

III. Pipeline decomposition of speech decoders

The principle of today's speech decoders is to find the most probable word sequences $\hat{W}$ given a sequence of features O that are extracted from speech waveform as shown in Equation (1).

$$\hat{W} = \underset{W}{\operatorname{argmax}} P(W|O) \tag{1}$$

Features are typically represented by a 20 to 50 dimensional real-valued vector that corresponds to a 10 ms time frame. For example, 10 second length utterance is represented by a sequence of 1000 feature vectors. At this top level, the corresponding pipeline is depicted as shown in Figure 5. In the figure, "*Feature Extraction*" is a function that takes a waveform signal as a list and returns a list O that represents a sequence of feature vectors. The argmax operation is performed by the "*search*" function that takes the list O and returns a list that contains the resulting word sequence W.

A popular choice for the features is the Mel-frequency cepstrum [11]. The first step of the extraction is to segment X by a sliding window with a shift interval equal to the frame rate (e.g. 10ms), which makes a sequence of speech vectors. Then, Fourier transform, filter-banking, log-magnitude transform, and discrete cosine transform (DCT) are sequentially applied. This process is within the territory of traditional signal processing and it is represented by a linear pipeline as shown in Figure 6. The front-end processor of the Sphinx decoder [2] is designed as a sequence of replaceable blocks which can be seen as a pipeline of modules.

The search function is the heart of decoders. For the $argmax$ operation, various possibilities of word sequences must be tested for the input feature sequence O. The probability score $P(W|O)$ is computed based on an acoustic model and a language model represented by HMM and N-gram. Since the possible number of word sequences is exponential to their length, efficient search strategies are required. The recent trend is to use weighted finite state transducers (WFSTs) [12]. HMM state transitions, N-grams, and other models used in decoders can be represented as WFSTs, and once they are represented by WFSTs, they can be composed to a single WFST.

A WFST consists of states and arcs where each arc has input/output labels and a weight. Two special states are a start and a final states. The search starts from the start state and ends in the final state. With WFST, finding the best word sequence is reduce to a problem of finding a path that starts from the start state and ends in the final state having the highest probability score. The input label of the arc corresponds to a HMM state name. Each transition for an arc having non-epsilon input label consumes a time frame of feature vectors, and the acoustic score is computed by the associated HMM state distribution and the feature frame. The computed acoustic score is merged with the language score embedded in the arc weight. The path score is the accumulation of the arc score along the path[1].

The search can be efficiently performed by using dynamic programming. First as an initialization, arcs from the start state with epsilon input labels are expanded obtaining an array of partial paths with accumulated scores. From the array, an array of states of the path ends is obtained with the accumulated scores, which is used as an initial set of states for the next expansion. After the initialization, expansions for non-epsilon transitions and for epsilon transitions are repeated in turn. For the non-epsilon transition, a feature frame is consumed and an array of partial paths are obtained. If the same state is reached from multiple states at the same timing, the one with the best accumulated scores is retained and others are discarded. To reduce the computational cost, the state array can be pruned so that states with the worst scores are removed based on a threshold. The process is call Viterbi-beam search, and we show the pipeline representation in Figure 7. The output H of this pipeline is a list representing a sequence of partial path arrays with the accumulated path scores, where each partial path array corresponds to a time frame.

To obtain a word sequence from the sequence of partial path arrays H, a back-tracking from the last element to the first is needed. However, this causes a problem for the pipeline based implementation since it can not be started until the end of O is processed. This means the process needs memory linear to the length of the input. It also means that the output is obtained only after the input ends. This may be Ok if the input is cut

[1]It is a product if the score is probability and a sum if it is log probability.

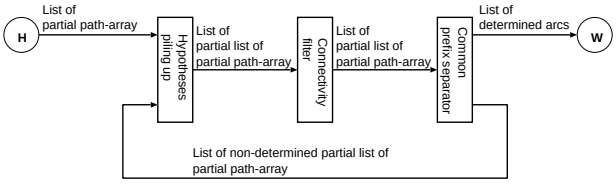

Fig. 8. Pipeline for progressive backtracking.

beforehand to shorter segments. Otherwise, long or infinite inputs are not acceptable.

The problem is solved by adopting live decoding or progressive back-tracking as used in some existing decoders [3]. In the progressive back-tracking, partial path arrays in H are piled up concatenated with a list of non-determined partial path arrays sent from the previous step. Then, the connectivity of the partial path arrays to the currently active hypotheses nodes are inspected, and those non-connected are removed. After the connectivity filtering, the common prefix of the partial list of partial path arrays is separated from the remaining part. The common prefix is a sequence of arcs of the determined part of the recognition result. The remaining non-determined part is sent back to the piling process. The pipeline of this process is shown in Figure 8. The output W is the sequence of the recognized words. The words are output progressively without waiting for the end of the input.

IV. Experimental setup

Based on the pipeline decomposition, a newer version of Husky decoder "Husky2" is designed and implemented with Haskell. Recognition experiments were performed using the standard evaluation set of the Corpus of Spontaneous Japanese (CSJ) consisting of 10 academic presentations [13]. The length of each presentation is about 10 to 20 minutes and the total duration is 2.3 hours. Triphone HMM with 3000 states and 3-gram language model with 30k vocabulary were made from the training set of CSJ. The Husky2 decoder takes a list of feature files and a WFST composed from the HMM and the N-gram as the input, and outputs recognition results. The details of the setup are the same as our previous paper [8].

V. Results

Table I shows recognition results in terms of word accuracy, memory size, and real time factor when manually segmented utterances are input. Compared to the previous version of our Husky decoder, Husky2 requires much smaller amount of memory and CPU time. The memory requirement is still large, but the recognition accuracy outperformed the Julius decoder with smaller RTF. Since Husky2 supports the progressive back-tracking, recognition experiments without the manual segmentation were also performed inputting the long lectures as they were. The word accuracy in this condition was 80.7%, which was closed to the one with the manually segmented inputs. The increase of memory and RTF were relatively small and they were 10.9 Gbyte and 6.9, respectively.

TABLE I
Word accuracy (WAcc), memory size, and Real time factor (RTF) by several decoders. Real time factor (RTF) is ratio of CPU time and length of input speech.

Decoder	WAcc	Memory (bytes)	RTF
Julius	79.8%	400 M	8
T^3	81.2%	400 M	3
Husky (old)	81.1%	40 G	62
Husky2 (new)	81.1%	9.6G	6

VI. Conclusions

Seeking for a new programming framework that efficiently describes speech recognition systems, we have proposed to describe speech decoders as pipelines and have shown how they are decomposed to them. Once they are described as pipelines, they are efficiently implemented by pure functional programming languages with delayed evaluation. Experimental results demonstrated that the pipeline based design is useful to implement efficient codes with a pure functional language. Future work includes reducing the memory cost. While part of the memory cost is an expense for comfortable automatic garbage collection, it could be largely reduced by improving the internal WFST and hypotheses representations.

Acknowledgments

This research has been supported by a grant from the Telecommunications Advancement Foundation (TAF). Part of this research has also been supported by KAKENHI (21300066), KAKENHI (23650068), and KAKENHI(21300060).

References

[1] A. Lee, T. Kawahara, and S. Doshita, "An efficient two-pass search algorithm using word trellis index," in *Proc. ICSLP*, 1998, pp. 1831–1834.

[2] Paul Lamere, Philip Kwok, William Walker, Evandro B. Gouvêa, Rita Singh, Bhiksha Raj, and Peter Wolf, "Design of the CMU sphinx-4 decoder," in *Proc. Interspeech*, 2003, pp. 1181–11184.

[3] P. R. Dixon, D. A. Caseiro, T. Oonishi, and S. Furui, "The titech large vocabulary wfst speech recognition system," in *Proc. IEEE ASRU*, 2007, pp. 443–448.

[4] MathWorks, *SIMULINK dynamic system simulation for MATLAB user's guide*.

[5] Maija Marttila Kontio, *Visual data flow programming languages: challenges and opportunities*, Ph.D. thesis, University of Eastern Finland, 2011.

[6] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, 1989.

[7] Hideki John Reekie, *Realtime Signal Processing: Dataflow, Visual, and Functional Programming*, Ph.D. thesis, University of Technology at Sydney, 1995.

[8] Takahiro Shinozaki, Masakazu Sekijima, Shigeki Hagihara, and Sadaoki Furui, "A compact speech decoder based on pure functional programming," in *Proc. APSIPA*, 2011.

[9] S. Thompson, *Haskell : the craft of functional programming*, Addison Wesley, March 1999.

[10] R. Plasmeijer and M. V. Eekelen, *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[11] S.B. Davis and P. Mermelstein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences," *IEEE Transaction on Acoustic Speech and Singal Processing*, vol. 28, no. 4, pp. 357–366, 1980.

[12] Mehryar Mohri, Fernando Pereira, and Michael Riley, "Weighted finite-state transducers in speech recognition," *Computer Speech and Language*, vol. 16, no. 1, pp. 69 – 88, 2002.

[13] T. Kawahara, H. Nanjo, T. Shinozaki, and S. Furui, "Benchmark test for speech recognition using the Corpus of Spontaneous Japanese," in *Proc. SSPR2003*, 2003, pp. 135–138.