Contents

List of Figures

List of Tables

Glossary

ALM	Adaptive logic module
ANN	Artificial neural network
API	Application programming interface
ASR	Automatic speech recognition
CD	context-dependent
CI	context-independent
CMN	Cepstral mean normalization
CMVN	Cepstral mean variance normalization
DTW	Dynamic time warping
EM	Expectation-maximization
FPGA	Field programmable gate arrays
GMM	Gaussian mixture model
HDL	Hardware description language
HMM	Hidden Markov model
IP	Intellectual property
LM	Language modelling
LPC	Linear predictive coding
MFCC	Mel frequency cepstral coefficient
MLP	Multi-layer perceptrons
PLP	Perceptual linear prediction
RTF	Real-time factor
SOPC	System-on-a-programmable-chip
VHDL	VHSIC hardware description language
VHSIC	Very-high-speed integrated circuits
WER	Word error rate
WFST	Weighted finite state transducer

1 Introduction

1.1 Objectives of the thesis

Automatic speech recognition (ASR) is the process of converting a human speech signal to a sequence of words by a computer. The use of ASR offers another path of communication between humans and computers.

Despite many research efforts, speech recognition has not been widely used in embedded systems. One of the reasons for this is that embedded platforms are limited in terms of processing power and computing resources [45, 49]. In comparison with common desktop processors, embedded systems generally have the following limitations.

- Slower CPU clock rate. It increases the execution time for running a program.
- Limited memory bandwidth. The speed of transferring data between memory and a CPU is generally slower in embedded systems. It increases the time required for accessing data from memory, which results in long periods of pipeline stalls.
- Limited CPU cache size. Compared with a desktop processor, the amount of caching resources inside an embedded CPU is limited. This often leads to a high cache miss rate, which increases the frequency of accessing data from memory.
- Lack of hardware support for floating-point operations. Algorithms may need to be modified to work with limited computing resources.

These limiting factors contribute to an increase of the decoding time. When embedded ASR is utilized as a human-machine interface, long response time can lead to poor user experiences. Therefore, a compromise has to be made between decoding speed and system performance. One possible way is to reduce the size of the vocabulary. As the number of words in the vocabulary decreases, the size of the search space is reduced and thereby decreasing the number of active hypotheses in the search space.

Another way is to use less precise data and/or coarser speech models. For example, fixed-point arithmetic is used instead of floating-point arithmetic, or fewer parameters are used for acoustic modelling and language modelling. These approximations can shorten the decoding time; however, they may also degrade the system performance in terms of recognition accuracy.

The first objective of the thesis is to develop a system architecture that is suitable for embedded ASR applications. In order to achieve real-time performance, we focus on the design of the architecture rather than other approaches that will degrade system performance in terms of recognition accuracy and vocabulary size. An overview of different system architectures is presented in Section 1.2. Our target application is a 1000-word command-and-control task.

Apart from recognition accuracy, decoding speed and vocabulary size, another point of consideration when building a practical ASR application is the adaptability of the system. Typically, an ASR system performs speech recognition based on a group of knowledge sources that are trained from a set of training data. These knowledge sources include but do not limit to acoustic models, context-dependency knowledge, lexical knowledge and language models. One way to utilize these knowledge sources is to combine them into a *static* search space. The ASR system then searches for the most probable sequence of words from the search space. This is the most efficient way for search space representation in terms of decoding speed, since the ASR system does not require to combine the knowledge sources during decoding.

However, static search space representation is not flexible since it does not allow on-line adaptation of the knowledge sources. It is difficult to modify the knowledge sources once they have been combined into a unified search space. In practice, the knowledge sources have to adapt to various types of situations during the operation of the ASR system [45]. For example, in a dialogue system, the language model (grammar) needs to change in response to different dialog states. Also, users may want to modify the dialog system by adjusting the grammar and/or introducing new words to the lexicon. Besides user customization, an ASR system can also automatically update the underlying knowledge sources for better recognition accuracy. One example is the adaptation of language model probabilities based on the topic of the current conversation.

The second objective of the thesis is to develop an algorithm which can *dynamically* integrate different kinds of knowledge sources during decoding. In this approach, the knowledge source that is subject to adaptation is factorized from the entire search space. By doing this, the knowledge source can be modified and adapted to the current working environment. The adapted knowledge source is then combined with the others during decoding. In this thesis, our proposed algorithm will be compared with other dynamic algorithms in terms of recognition accuracy and decoding speed.

1.2 Current embedded ASR system architectures

In many embedded ASR applications, researchers and developers have been focussing on building a *pure software-based* implementation of the recognition engine on an embedded platform [26, 9]. As the name suggests, a pure software-based system is a platform on which developers describe the speech recognition algorithm in programming languages. The source code is compiled and then assembled into the machine code of the target processor. This approach enables fast development of ASR applications. Developers can directly port the source code from a desktop environment onto an embedded platform without any major modifications. However, this pure software-based approach cannot take the advantage of parallelism adhered to the speech recognition algorithm. Therefore, they often need algorithmic modification and approximation in order to meet the real-time requirement, which leads to degradation in recognition accuracy.

At another extreme, some researchers focus on building a *pure hardware-based* system [32, 33, 57]. In these systems, the entire recognition engine is built in the form of digital circuits. The recognition engine is often described in hardware description language (HDL), which can be synthesized on some target technologies, for example, a field programmable gate array (FPGA) device. While parallelism and co-processing can be fully employed in these systems, the development time is often more than the pure software-based systems. In addition, it is more difficult to modify and re-structure the design in this approach.

As a compromise, a *hardware-software co-design* approach seems to be attractive [31, 62]. A simple hardware-software co-processing system consists of a processor and one or more hardware functional units. In this approach, developers identify the computationally intensive parts of the algorithm. Hardware functional units, which are described in HDL, are built to employ possible algorithmic parallelism in the computationally intensive parts of the algorithm. At the same time, software developers can write source code for the processor, which controls the hardware functional units and runs the rest of the algorithm. This approach combines the advantages of the pure software-based and the pure hardware-based approaches. Hardware-software co-design provides:

- Rapid prototyping of applications. The instruction set of the processor in a co-processing system provides abstraction of the underlying architecture. Without knowing every detail of the underlying architecture, developers build their applications in software. It shortens the development time and cost compared with the pure hardware-based approach.

- Faster processing time. A hardware-software co-processing system employs parallelism in the following ways.
  - Hardware functional units accelerate the computationally intensive parts of the algorithm by performing independent operations in parallel.
  - The entire algorithm is divided into independent sub-tasks. Each sub-task is handled by a hardware functional unit. The sub-tasks are performed in parallel.
- Flexibility in design modification. It is generally more difficult to modify an algorithm in the pure hardware-based approach since it deals with low-level structures, such as registers and logic gates. At the other end of the spectrum, it is much easier to make modification of the algorithm in its software form because the details of the underlying architecture is hidden away from the developers. The hardware-software co-processing approach is in the middle of the spectrum. While the invariant operations within the algorithm can be implemented in hardware, the part which may need future modification can be implemented in software. It allows flexibility in design modification and future enhancement.
- Extensibility from one application to a group of applications. Developers can specialize in building their application in a hardware functional unit. Together with its software application programming interface (API), the functional unit can be developed as an intellectual property (IP) block, which can be re-used and integrated with other IP blocks for building a useful group of applications. For example, a speech-controlled intelligent environment can be built by incorporating a speech recognition engine with various electrical appliance controllers.

In this thesis, we adopt the hardware-software co-design approach to develop an embedded speech recognition system. We analyze the speech recognition algorithm and identify where the computational bottlenecks are. We build hardware functional units in order to accelerate the critical parts of the algorithm. Finally, the proposed system is assessed by word accuracy rate and real-time factor.

1.3 Contributions of the thesis

The first objective of this thesis is to develop an embedded ASR system that is suitable for real-time applications. This part of the work includes the following.

- A hardware-software co-processing ASR system is proposed. With the same level of word accuracy, the proposed system shows a better timing performance than other reported systems. In comparison with [31, 62], which is also a hardware-software co-processing approach, the architecture of the proposed system is more portable to larger vocabulary tasks and different conversation topics. In addition, we present a more detailed timing analysis showing the real-time factor of the system. On the contrary, this timing measure is not clearly shown in [31, 62].

- An adaptive beam pruning algorithm, which limits the number of active hypotheses, is proposed. It further improves the timing performance of the co-processing system. In terms of implementation, the proposed algorithm is easier compared with other similar pruning technique, such as histogram pruning [58].

- A framework for converting various data types from floating-point to fixed-point is proposed. The decoding speed is much faster without any degradation in recognition accuracy when the fixed-point implementation of the recognizer is tested on the target embedded platform. The proposed framework is a complete framework covering all the data types in the ASR algorithm. Other studies focus on only a certain part of the algorithm [22, 28, 30].

The second objective is to develop an algorithm which can dynamically integrate different kinds of knowledge sources during decoding. In this thesis, each knowledge source is represented by a weighted finite state transducer (WFST). This part of the work includes the following.

- A generalized dynamic WFST composition algorithm is proposed. Compared with other dynamic composition approaches, the proposed algorithm avoids the creation of non-coaccessible transitions, performs weight look-ahead and does not impose any constraints to the topology of the WFSTs. Experimental results show that the proposed algorithm shows a better word accuracy versus real-time characteristics than those of the other approaches.

1.4 Outline of the thesis

Chapter 2 gives an overview of the fundamentals of automatic speech recognition (ASR). The components of a typical ASR system, such as feature extraction, acoustic modelling and language modelling, are described. Different types of search space representations and search algorithms are also discussed.

Chapter 3 introduced a speech recognizer called Juicer, which uses weighted finite state transducer (WFST) for search space representation. Juicer is the underlying speech recognition engine of our embedded ASR system. In this chapter, the theory of WFST is described. It also explains how each knowledge source is represented by a WFST and how all the WFSTs are combined into a unified search space statically.

Chapter 4 to Chapter 6 describe the development process of the embedded ASR system. In Chapter 4, a framework for converting different types of data from floating-point to fixed-point is proposed. In the experiments, we start with a floating-point implementation of Juicer and derive a fixed-point version based on the proposed framework.

Chapter 5 describes the baseline system that is used for benchmarking purposes. The baseline system is a pure software-based ASR system with an Altera Nios II embedded processor implemented on a Stratix II FPGA device, which serves as the target embedded platform. The timing performances of both the floating-point implementation and the fixed-point implementation are examined on this platform. The computational bottleneck of the ASR algorithm is also identified.

In Chapter 6, the system architecture of a hardware-software co-processing ASR system is proposed. The timing performance of the proposed system is compared with that of the pure software-based baseline system as described in Chapter 5. In addition, an adaptive beam pruning algorithm is also introduced, which further improves the timing performance. At the end of this chapter, a comparison between the proposed system and other reported systems is presented.

In Chapter 7, a generalized dynamic WFST composition algorithm is proposed. The pseudocode of the algorithm is presented. Experimental results are also shown for comparison with other dynamic composition approaches.

Chapter 8 summarizes the work and presents the conclusions of the thesis.

2 Fundamentals of speech recognition

2.1 The decoding problem

The goal of automatic speech recognition (ASR) is to find the most probable word sequence given that a sequence of acoustic feature vectors is observed. This decoding process can be expressed mathematically by the following equation,

$$\hat{W} = \underset{W_1^n}{\operatorname{argmax}}\{P(W_1^n|\mathbf{o}_1^T, \Theta)\} \tag{2.1}$$

where Θ are the parameters of the ASR system. The equation denotes that given a sequence of T acoustic feature vectors, $\mathbf{o}_1^T = \{\mathbf{o}_1, \mathbf{o}_2, ..., \mathbf{o}_T\}$, the output of the recognizer is a sequence of n words, W_1^n, which gives the maximum posterior probability, $P(W_1^n|\mathbf{o}_1^T, \Theta)$. For isolated word ASR tasks, the size of the vocabulary is small. Thus, direct calculation of (2.1) is trivial. However, for larger vocabulary tasks, the calculation becomes intractable since all the possible word sequences need to be considered. In order to make ASR feasible, the equation is rewritten as follows using Bayes' rule.

$$\begin{aligned} \hat{W} &= \underset{W_1^n}{\operatorname{argmax}}\left\{\frac{P(\mathbf{o}_1^T|W_1^n, \Theta_a)P(W_1^n|\Theta_l)}{P(\mathbf{o}_1^T|\Theta)}\right\} \\ &= \underset{W_1^n}{\operatorname{argmax}}\{P(\mathbf{o}_1^T|W_1^n, \Theta_a)P(W_1^n|\Theta_l)\} \end{aligned} \tag{2.2}$$

Since $P(\mathbf{o}_1^T|\Theta)$ is independent of W_1^n, it is omitted in the equation. Θ_a is referred to as a set of parameters known as the *acoustic model*. Θ_l comprises the parameters of the *language model*. An acoustic model describes the probabilistic relationship between the acoustic feature vectors and the words, while a language model contains the linguistic information between words.

Figure 2.1 illustrates the block diagram of a typical ASR system. *Feature extraction* converts the speech samples to a sequence of acoustic feature vectors. The characteristics of the underlying speech signal are captured by the feature vectors. Given the feature vectors are observed, the *decoder* finds the most probable word sequence based on the information from the acoustic model and the language model.

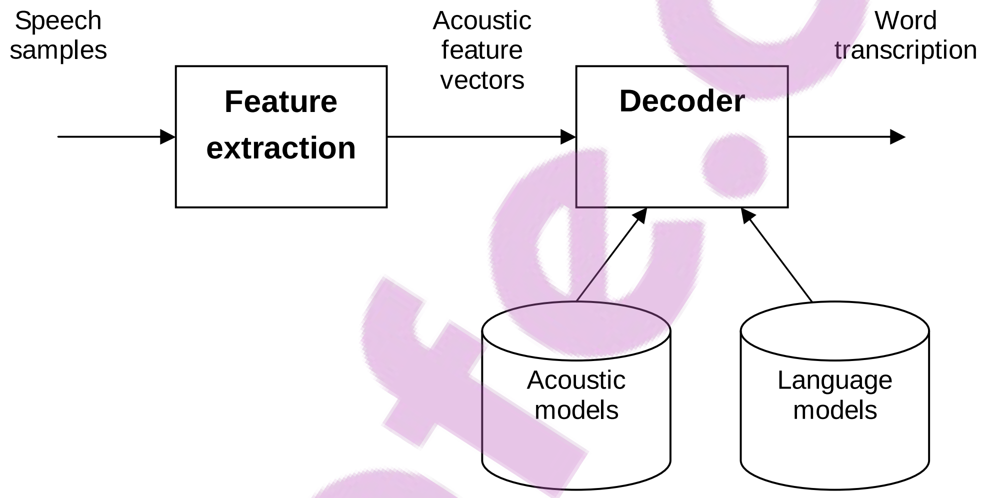

Figure 2.1: Block diagram of a typical ASR system

2.2 Feature extraction

The purpose of feature extraction is to provide the ASR system with a sequence of acoustic feature vectors. The first step of feature extraction is *pre-emphasis*. Usually, more energy is concentrated at the low-frequency portion of a speech spectrum than the high-frequency portion. The idea behind pre-emphasis is to emphasize the high-frequency portion of the speech spectrum. Pre-emphasis is often realized by a first-order high-pass finite impulse response (FIR) filter. Its transfer function in z-domain is $H(z) = 1.0 - \alpha z^{-1}$, where α is a constant with a typical value of 0.97.

Speech signal is non-stationary. Therefore, in order to analyze its characteristics, a speech signal is segmented into a series of shorter speech signals called *frames*, in which quasi-stationary is assumed. Each frame is essentially a window extracted from the speech signal. Frame duration is typically 25ms. An acoustic feature vector consisting of a number of coefficients is extracted from a speech frame. After extracting a feature vector, the frame is proceeded by a certain amount of time, typically 10ms, for the next feature vector. Feature extraction continues until the frame reaches the end of the speech signal.

The choice of the window function is important. It is because windowing is equivalent to multiplying the speech signal by the window function. In frequency domain, the speech spectrum is convolved with the frequency response of the window function. In order to reduce spectral leakage, a Hamming window is often used instead of a rectangular window since the side lobes of the frequency response of a Hamming window have smaller magnitudes than those of a rectangular window [55].

After Hamming-windowing, the speech frame is converted to an acoustic feature vector. There are many different feature extraction algorithms. Some of the commonly-used algorithms include mel frequency cepstral coefficients (MFCC) [17], perceptual linear prediction (PLP) cepstral coefficients [23] and linear predictive coding (LPC) [34, 55] coefficients. The following paragraphs give a brief overview of these feature extraction algorithms.

MFCC First, the power spectrum (that is, the magnitude spectrum squared) of the speech frame is found. Then, the power spectrum is filtered by a series of triangular-shaped bandpass filters. These filters are spaced uniformly on a non-linear frequency scale called the mel scale, which is a psychoacoustic measure of pitches judged by human [59]. The mel scale is approximately linear to the linear frequency scale below 1 kHz and logarithmic above 1 kHz. After the mel filter bank, the filter outputs are compressed by the logarithmic function. Finally, the log filter outputs are decorrelated by discrete cosine transform (DCT).

PLP Like MFCC, the first step is to find the power spectrum of the speech frame. Then the power spectrum is filtered by a series of trapezoidal-shaped bandpass filters, which are spaced uniformly on a non-linear frequency scale called the Bark scale. The centre frequencies of these filters are approximately 1 Bark apart [23]. Unlike MFCC where pre-emphasis is done before windowing, the Bark filter outputs are weighed by the equal-loudness curve. After that, the filter outputs are compressed by the cubic root function. Next, inverse discrete Fourier transform (IDFT) is applied to the compressed filter outputs and the resultant sequence is treated as the autocorrelation coefficients. The rest of the algorithm is the same as the LPC method as discussed later. Linear predictive coefficients, which can be subsequently converted to cepstral coefficients by a simple recursion, are found using the autocorrelation method [55].

LPC In contrast with MFCC and PLP, the LPC method does not find the spectrum of the speech frame. Instead, the autocorrelation coefficients of the speech frame are found. Using the autocorrelation method [55], a set of LPC coefficients is generated from the autocorrelation coefficients. The LPC coefficients are the coefficients of an all-pole digital filter, which models the transfer function of the human vocal tract.

Typically, each feature vector consists of 12-14 coefficients. These coefficients are known as *static* coefficients. In order to model the time correlation between the feature vectors, the first and the second order temporal derivatives of the feature vectors are often used [21]. The time trajectory of each static coefficient is modelled by a second order polynomial [54]. The first order temporal derivatives (also known as the *delta* features) and the second order temporal derivatives (also known as the *delta-delta* features) are found by differentiating the regression polynomial. The static features, together with the delta and delta-delta features, are concatenated to form a single feature vector.

To reduce the feature variability in different recording and testing environments, either cepstral mean normalization (CMN) [5] or cepstral mean variance normalization (CMVN) [15] is applied to the feature vectors. In CMN, the mean of each coefficient is normalized to 0. In CMVN, the mean and the variance are normalized to 0 and 1 respectively.

2.3 Acoustic modelling

The purpose of acoustic modelling is to provide a framework for describing the relationship between a sequence of acoustic feature vectors, $\mathbf{o}_1^T$, and a sequence of words, W_1^n. In early ASR research, template-matching technique, for example, dynamic time warping (DTW) [56], was one of the widely-used acoustic models. However, when the recognition task becomes more complicated, it requires to store a large number of templates, which is infeasible for practical applications. Besides, it is difficult to use DTW to generalize unseen data. Hence, stochastic modelling, for example, hidden Markov model (HMM), becomes one of the most commonly-used approaches for current ASR systems.

2.3.1 Hidden Markov Model (HMM)

The topology of an HMM is a finite state machine (FSM). Figure 2.2 illustrates an HMM example. It consists of a number of states and transitions. An HMM can be viewed as a double-embedded stochastic process [53]. The first stochastic process is the probabilistic model of the observations. In ASR, an acoustic feature vector is an observation. We *know* (observe) what the feature vector is from feature extraction. A probabilistic model of the observation is associated with each HMM state. The second stochastic process is the model of the state sequence. State transitions are modelled by a probabilistic function associated with the current state. State sequences are not observable, therefore they are *hidden*. We *do not know* what the state sequences are. However, some probabilistic sense of the underlying state sequence can be derived from the observation sequence.

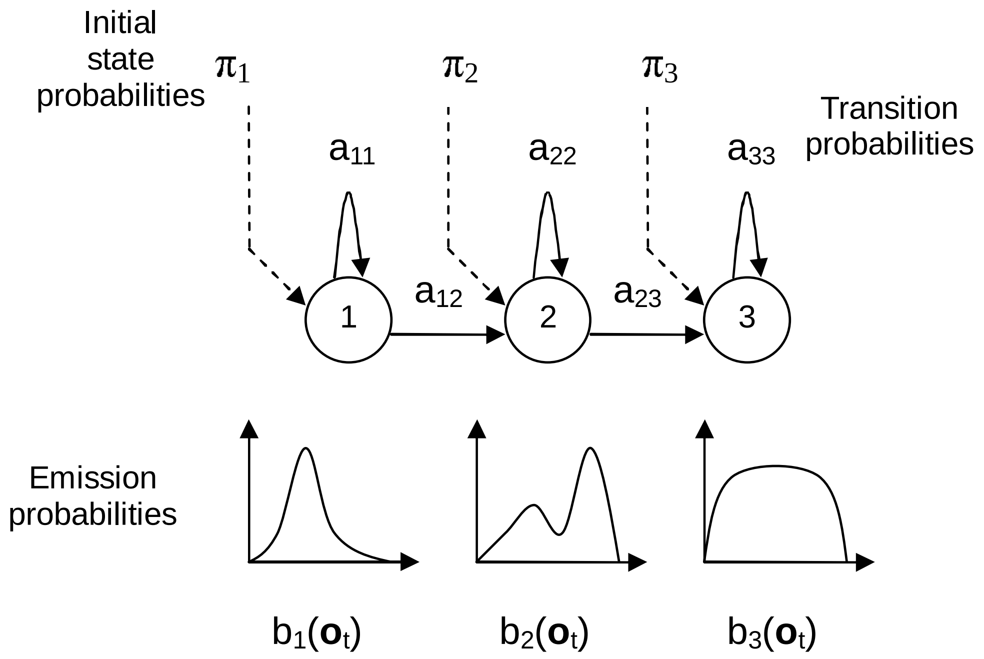

Figure 2.2: The topology of an HMM example. In this example, there are 3 states. The probabilities of being in an HMM state at the start of a state sequence are called the *initial state probabilities*. State transitions are modelled by *transition probabilities*. The probabilities of the observation vectors are known as *emission probabilities*. Each HMM state has an associated emission probability distribution.

An HMM can be defined by the following elements.

- $\mathbf{O}$ - The observation alphabet. In most cases, continuous feature vectors are used. However, if discrete feature vectors are needed, vector quantization can be used to map a continuous feature vector to a discrete symbol.

- Ω - The number of HMM states in the model.

- $A = \{a_{ij}\}$ - A transition probability matrix, where a_{ij} denotes the probability of taking a transition from state i (at time t) to state j at (at time $t+1$).

$$a_{ij} = P(q_{t+1} = j | q_t = i) \tag{2.3}$$

- $B = \{b_j(\mathbf{o}_t)\}$ - The emission probability matrix. This is the probabilistic model of the observation associated with each HMM state. $b_j(\mathbf{o}_t)$ is the probability of the observation given the HMM state is j at time t.

$$b_j(\mathbf{o}_t) = P(\mathbf{o}_t | q_t = j) \tag{2.4}$$

- $\boldsymbol{\pi} = \{\pi_j\}$ - The initial state probability distribution.

$$\pi_j = P(q_1 = j) \tag{2.5}$$

As previously mentioned in this section, an acoustic model describes the probabilistic relationship between a sequence of features vectors and a sequence of words. Mathematically, this probabilistic relationship is denoted by $P(\mathbf{o}_1^T|W_1^n, \Theta_a)$ as shown in (2.2). HMM can be used to build the acoustic model for describing this relationship. In early ASR research, each word is represented by an HMM. The acoustic model of a word sequence is built by concatenating the word HMMs together. Training of a word HMM requires a sufficient number of word instances in the training data. Therefore, this approach becomes infeasible when the size of the vocabulary increases. As a result, sub-word HMMs are introduced for acoustic modelling. Here, each word is treated as a concatenation of some sub-word units, such as, phonemes, and each sub-word unit is modelled by an HMM. For example, the word "cat" is a concatenation of three sub-word units: /k/, /ae/ and /t/. These sub-word units are called *context-independent* sub-word units since these units are trained without considering their surrounding sub-word units. The /ae/ unit in "cat" is grouped in the same class as the /ae/ unit in the word "man".

In order to model coarticulation between surrounding sub-word units, *context-dependent* sub-word units are used. In this approach, the class label of a sub-word unit takes into account of its surrounding sub-word units. For example, the /ae/ phoneme of the word "cat" is assigned to the /k−ae+t/ class where /k/ and /t/ are the left and the right contexts respectively. Similarly, the /ae/ phoneme of "man" is given another label /m−ae+n/. These sub-word units are also known as *triphones*. Some phonemes can be modelled with their left or right context only. For example, the /k/ phoneme of "cat" is modelled with the /ae/ phoneme only. A *biphone* label /k+ae/ is given to this sub-word unit. Hence, the word "cat" is a concatenation of two biphones and one triphone: /k+ae/, /k−ae+t/ and /ae−t/. Each context-dependent sub-word unit is modelled by an HMM.

Given that each word is modelled by a series of sub-word HMMs, a word sequence is essentially a sequence of HMM states. Thus, $P(\mathbf{o}_1^T|W_1^n, \Theta_a)$ can be rewritten as follows.

$$P(\mathbf{o}_1^T|W_1^n, \Theta_a) = \sum_{q_1^T} P(\mathbf{o}_1^T, q_1^T|W_1^n, \Theta_a) \tag{2.6}$$

where q_1^T is one of the possible HMM state sequences representing the word sequence W_1^n. The evaluation of (2.6) is computationally expensive. Hence, it can be approximated under the Viterbi criterion.

$$P(\mathbf{o}_1^T|W_1^n, \Theta_a) \approx \max_{q_1^T} P(\mathbf{o}_1^T, q_1^T|W_1^n, \Theta_a) \tag{2.7}$$

where summation is replaced by the max operation. In order to use (2.6) and (2.7) for ASR, the following three basic problems need to be solved [53]:

1. **Evaluation**. Given the observation sequence $\mathbf{o}_1^T$, and the acoustic model Θ_a, how

to evaluate $P(\mathbf{o}_1^T|W_1^n,\Theta_a)$?

2. **Decoding**. Given the observation sequence $\mathbf{o}_1^T$, and the acoustic model Θ_a, how to find the optimal HMM state sequence q_1^T, that maximizes $P(\mathbf{o}_1^T, q_1^T|W_1^n,\Theta_a)$?

3. **Training**. Given a set of training data, how to determine the parameters of the acoustic model, $\Theta_a = (A, B, \boldsymbol{\pi})$, which maximize the likelihood of the training data?

The following sections describe briefly the solutions of the above three problems.

2.3.2 Evaluation of HMM

In this problem, we would like to evaluate $P(\mathbf{o}_1^T|W_1^n,\Theta_a)$ according to (2.6). Equation (2.6) can be expressed as follows.

$$\begin{aligned} P(\mathbf{o}_1^T|W_1^n,\Theta_a) &= \sum_{q_1^T} P(\mathbf{o}_1^T, q_1^T|W_1^n,\Theta_a) \\ &= \sum_{q_1^T} P(\mathbf{o}_1^T|q_1^T, W_1^n,\Theta_a)P(q_1^T|W_1^n,\Theta_a) \end{aligned} \tag{2.8}$$

To simplify the evaluation of the above equation, two assumptions are made. First, it is assumed that the Markov chain is first-order. In other words, a state transition only depends on the preceding state. Originally, $P(q_1^T|W_1^n,\Theta_a)$ in (2.8) is defined as:

$$P(q_1^T|W_1^n,\Theta_a) = P(q_1|W_1^n,\Theta_a)\prod_{t=2}^{T} P(q_t|q_1^{t-1}, W_1^n,\Theta_a) \tag{2.9}$$

With the first-order Markov chain assumption, the equation can be rewritten as follows.

$$\begin{aligned} P(q_1^T|W_1^n,\Theta_a) &= P(q_1|W_1^n,\Theta_a)\prod_{t=2}^{T} P(q_t|q_{t-1}, W_1^n,\Theta_a) \\ &= \pi_{q_1}\prod_{t=2}^{T} a_{q_{t-1},q_t} \end{aligned} \tag{2.10}$$

The second assumption is that an observation only depends on the current HMM state. It is independent of other observations and other HMM states. Hence, $P(\mathbf{o}_1^T|q_1^T, W_1^n,\Theta_a)$

in (2.8) can be simplified as follows.

$$\begin{aligned} P(\mathbf{o}_1^T|q_1^T, W_1^n, \Theta_a) &= \prod_{t=1}^{T} P(\mathbf{o}_t|q_t, W_1^n, \Theta_a) \\ &= \prod_{t=1}^{T} b_{q_t}(\mathbf{o}_t) \end{aligned} \tag{2.11}$$

In other words, using (2.10) and (2.11), Equation (2.8) becomes:

$$P(\mathbf{o}_1^T|W_1^n, \Theta_a) = \sum_{q_1^T} \pi_{q_1} \prod_{t=2}^{T} a_{q_{t-1},q_t} \prod_{t=1}^{T} b_{q_t}(\mathbf{o}_t) \tag{2.12}$$

To efficiently evaluate (2.12), a recursive algorithm called *forward algorithm* is used. The *forward probability* is defined as follows.

$$\alpha_t(i) = P(\mathbf{o}_1^t, q_t = i|W_1^n, \Theta_a) \tag{2.13}$$

The forward algorithm has the following steps.

1. **Initialization**

$$\alpha_1(i) = \pi_i b_i(\mathbf{o}_1) \quad 1 \leq i \leq \Omega \tag{2.14}$$

2. **Induction** for $2 \leq t \leq T$

$$\alpha_t(j) = \left[\sum_{i=1}^{\Omega} \alpha_{t-1}(i) a_{ij} \right] b_j(\mathbf{o}_t) \quad 1 \leq j \leq \Omega \tag{2.15}$$

3. **Termination**

$$P(\mathbf{o}_1^T|W_1^n, \Theta_a) = \sum_{i=1}^{\Omega} \alpha_T(i) \tag{2.16}$$

where Ω is the number of states which constitute the HMM for modelling the word sequence W_1^n. The symbols i and j are the state indices of the HMM. The forward algorithm evaluates $P(\mathbf{o}_1^T|W_1^n, \Theta_a)$ according to (2.6).

2.3.3 Decoding of HMM

In Section 2.3.2, a forward algorithm is used to evaluate the full likelihood $P(\mathbf{o}_1^T|W_1^n, \Theta_a)$ as defined in (2.6). In this section, a *Viterbi algorithm* [60] is presented to find the approximation of $P(\mathbf{o}_1^T|W_1^n,\Theta_a)$ according to (2.7). Besides, an optimal HMM state sequence is also found which gives this approximation.

The major difference between (2.6) and (2.7) is that the summation operation in (2.6) is replaced by the max operation in (2.7). Like (2.6), the evaluation of (2.7) can be carried out by a recursive algorithm. Here, a quantity called *Viterbi likelihood* is defined as follows.

$$V_t(i) = \max_{q_1^{t-1}} P(\mathbf{o}_1^t, q_1^{t-1}, q_t = i|W_1^n,\Theta_a) \tag{2.17}$$

The Viterbi likelihood is the greatest probability of a state sequence, which ends in state i at time t and generates the first t observations. The Viterbi algorithm has the following steps. Another variable, $\Psi_t(i)$, is for recording the optimal state sequence.

1. **Initialization**

$$\begin{aligned} V_1(i) &= \pi_i b_i(\mathbf{o}_1) \quad 1 \le i \le \Omega \\ \Psi_1(i) &= 0 \end{aligned} \tag{2.18}$$

2. **Recursion** for $2 \le t \le T$

$$\begin{aligned} V_t(j) &= \left[\max_{1\le i\le\Omega} V_{t-1}(i)a_{ij}\right] b_j(\mathbf{o}_t) \quad 1 \le j \le \Omega \\ \Psi_t(j) &= \operatorname*{argmax}_{1\le i\le\Omega} V_{t-1}(i)a_{ij} \end{aligned} \tag{2.19}$$

3. **Termination**

$$\begin{aligned} P(\mathbf{o}_1^T|W_1^n,\Theta_a) &\approx \max_{q_1^T} P(\mathbf{o}_1^T, q_1^T|W_1^n,\Theta_a) = \max_{1\le i\le\Omega} V_T(i) \\ q_T^* &= \operatorname*{argmax}_{1\le i\le\Omega} V_T(i) \end{aligned} \tag{2.20}$$

4. **Backtracking of the optimal state sequence**

$$q_t^* = \Psi_{t+1}(q_{t+1}^*) \quad T-1 \ge t \ge 1 \tag{2.21}$$

In summary, the Viterbi algorithm finds the optimal state sequence $\{q_1^*, q_2^*, ..., q_T^*\}$ which maximizes the probability given the observation sequence $\mathbf{o}_1^T$. This probability approximates the full likelihood $P(\mathbf{o}_1^T|W_1^n,\Theta_a)$ according to (2.7).

2.3.4 Training of HMM

In this problem, given a set of U training utterances, the aim is to determine the parameters of the acoustic model $\Theta_a = (A, B, \boldsymbol{\pi})$ which maximizes the likelihood of the training utterances. One common method is to use *Baum-Welch algorithm*, which is also called the *forward-backward algorithm* [7]. It is the specific case of the expectation-maximization (EM) algorithm [18].

To use the forward-backward algorithm, we first define some helper variables. The first variable is the *backward probability*, which is defined as follows.

$$\beta_t(i) = P(\mathbf{o}_{t+1}^T | q_t = i, W_1^n, \Theta_a) \tag{2.22}$$

Similar to the forward probability, the backward probability can also be evaluated recursively.

1. **Initialization**

$$\beta_T(i) = 1 \quad 1 \leq i \leq \Omega \tag{2.23}$$

2. **Induction** for $T - 1 \geq t \geq 1$

$$\beta_t(i) = \sum_{j=1}^{\Omega} a_{ij} b_j(\mathbf{o}_{t+1}) \beta_{t+1}(j) \quad 1 \leq i \leq \Omega \tag{2.24}$$

3. **Termination**

$$P(\mathbf{o}_1^T | W_1^n, \Theta_a) = \sum_{j=1}^{\Omega} \pi_j b_j(\mathbf{o}_1) \beta_1(j) \tag{2.25}$$

Moreover, two more variables, $\gamma_t(i)$ and $\xi_t(i, j)$, are defined.

$$\begin{aligned} \gamma_t(i) &= P(q_t = i | \mathbf{o}_1^T, W_1^n, \Theta_a) \\ &= \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{\Omega} \alpha_t(j)\beta_t(j)} \end{aligned} \tag{2.26}$$

$$\begin{aligned} \xi_t(i, j) &= P(q_t = i, q_{t+1} = j | \mathbf{o}_1^T, W_1^n, \Theta_a) \\ &= \frac{\alpha_t(i) a_{ij} b_j(\mathbf{o}_{t+1}) \beta_{t+1}(j)}{\sum_{i=1}^{\Omega} \sum_{j=1}^{\Omega} \alpha_t(i) a_{ij} b_j(\mathbf{o}_{t+1}) \beta_{t+1}(j)} \end{aligned} \tag{2.27}$$

In other words, $\gamma_t(i)$ is the posterior probability of being in state i at time t, given the observation sequence $\mathbf{o}_1^T$. The variable $\xi_t(i, j)$ is the posterior probability of being in state i at time t and in state j at time $t + 1$, given the observation sequence. Using these two variables, the transition probabilities and the initial state distribution can be estimated as follows.

$$\bar{a}_{ij} = \frac{\sum_{u=1}^{U}\sum_{t=1}^{T^{(u)}-1}\xi_t^{(u)}(i,j)}{\sum_{u=1}^{U}\sum_{t=1}^{T^{(u)}-1}\gamma_t^{(u)}(i)} \tag{2.28}$$

$$\bar{\pi}_i = \frac{\sum_{u=1}^{U}\gamma_1^{(u)}(i)}{U} \tag{2.29}$$

In the equations, $\bar{a}_{ij}$ and $\bar{\pi}_i$ are the re-estimated parameters for one iteration. These parameters are used to re-calculate γ and ξ, which then re-estimate the parameters in the next iteration until convergence. The u superscript denotes the u^{th} utterance in the training data.

There are two major types of emission probabilities, $b_j(\mathbf{o}_t)$, in an HMM-based ASR system. These probabilities can be either modelled by Gaussian mixture models (GMM) or an artificial neural network (ANN). The training methods of $b_j(\mathbf{o}_t)$ are different in these two approaches. In the following two sections, we briefly describe these two HMM-based systems.

2.3.5 HMM/GMM system

In an HMM/GMM ASR system, the emission probability of an HMM state, $b_j(\mathbf{o}_t)$, is modelled by a mixture of Gaussian density functions.

$$\begin{aligned} b_j(\mathbf{o}_t) &= \sum_{m=1}^{M} c_{jm}\mathcal{N}(\mathbf{o}_t, \boldsymbol{\mu}_{jm}, \boldsymbol{\Sigma}_{jm}) \\ &= \sum_{m=1}^{M} c_{jm}\frac{1}{(2\pi)^{\frac{D}{2}}|\boldsymbol{\Sigma}_{jm}|^{\frac{1}{2}}}\exp\left[-\frac{1}{2}\left(\mathbf{o}_t-\boldsymbol{\mu}_{jm}\right)^T\boldsymbol{\Sigma}_{jm}^{-1}\left(\mathbf{o}_t-\boldsymbol{\mu}_{jm}\right)\right] \end{aligned} \tag{2.30}$$

where $\mathcal{N}$ is a Gaussian density. $\boldsymbol{\mu}_{jm}$ and $\boldsymbol{\Sigma}_{jm}$ are the mean vector and the covariance matrix of the m^{th} Gaussian respectively. D is the dimension of the observation vector. c_{jm} is the mixture weight of the m^{th} Gaussian with $\Sigma_{m=1}^{M}c_{jm} = 1$.

Training of the parameters can be done by using the forward-backward algorithm. Another variable is introduced for training the parameters.

$$\gamma_t(j,m) = \gamma_t(j)\left[\frac{c_{im}\mathcal{N}(\mathbf{o}_t, \boldsymbol{\mu}_{jm}, \boldsymbol{\Sigma}_{jm})}{\sum_{m=1}^{M} c_{jm}\mathcal{N}(\mathbf{o}_t, \boldsymbol{\mu}_{jm}, \boldsymbol{\Sigma}_{jm})}\right] \tag{2.31}$$

where $\gamma_t(j,m)$ is the probability of being at state j with the m^{th} mixture at time t given the observations. The parameters can be trained by evaluating the following equations until convergence [8].

$$\bar{c}_{jm} = \frac{\sum_{u=1}^{U}\sum_{t=1}^{T^{(u)}} \gamma_t^{(u)}(j,m)}{\sum_{u=1}^{U}\sum_{t=1}^{T^{(u)}} \gamma_t^{(u)}(j)} \tag{2.32}$$

$$\bar{\boldsymbol{\mu}}_{jm} = \frac{\sum_{u=1}^{U}\sum_{t=1}^{T^{(u)}} \gamma_t^{(u)}(j,m)\mathbf{o}_t^{(u)}}{\sum_{u=1}^{U}\sum_{t=1}^{T^{(u)}} \gamma_t^{(u)}(j,m)} \tag{2.33}$$

$$\bar{\boldsymbol{\Sigma}}_{jm} = \frac{\sum_{u=1}^{U}\sum_{t=1}^{T^{(u)}} \gamma_t^{(u)}(j,m)\left(\mathbf{o}_t^{(u)} - \boldsymbol{\mu}_{jm}\right)\left(\mathbf{o}_t^{(u)} - \boldsymbol{\mu}_{jm}\right)^T}{\sum_{u=1}^{U}\sum_{t=1}^{T^{(u)}} \gamma_t^{(u)}(j,m)} \tag{2.34}$$

The u superscript denotes the u^{th} utterance of the training corpus consisting of U utterances.

2.3.6 Hybrid HMM/ANN system

Another approach for modelling the emission probabilities of an HMM is to use an artificial neural network (ANN). One of the common ANN architectures used in ASR is multi-layer perceptrons (MLP) [10]. In an MLP, there is an input layer, one or more hidden layers and an output layer. Each layer consists of nodes, which are connected to the nodes in the next layer via weights. Each node is basically a computing unit. The following equation expresses the mathematical operation of the nodes in an arbitrary layer.

$$\mathbf{h} = F(\overline{\mathbf{W}}^T\overline{\mathbf{x}}) \tag{2.35}$$

In the equation, $\overline{\mathbf{x}} = (1, x_1, x_2, ..., x_d)^T$ is an input vector to the layer with an augmented value of 1. $\overline{\mathbf{W}} = \{w_{dk}\}$ is a $(d+1) \times K$ weight matrix and K is the number of nodes in the layer. $\mathbf{h} = (h_1, h_2, ..., h_K)^T$ is the output vector of which each element is the output of a node in the layer. $F(.)$ is a non-linear differentiable function. In practice, the non-linear function in the hidden layer is a sigmoid function which generates higher-order moments of the elements in the input vector, whereas in the output layer it is a softmax function which approximates the decision logic.

In ASR, several frames of acoustic feature vectors are concatenated to form an input vector. Typically, an input vector consists of the current frame (at time t) with 4 preceding and 4 succeeding feature vectors (that is, 9 frames of feature vectors). In the output layer, each node corresponds to one context-independent sub-word unit. After training the MLP, each node in the output layer estimates $P(q_t = k|\mathbf{o}_t)$, which is the posterior probability of the context-independent sub-word unit k given an observation vector $\mathbf{o}_t$ at the current time index t. In the hybrid HMM/ANN approach, each context-independent unit is modelled by an HMM state and q_t is the HMM state (context-independent unit) at time t.

To incorporate the MLP outputs into the HMM framework, Bayes' rule is applied to $P(q_t = k|\mathbf{o}_t)$.

$$\frac{P(\mathbf{o}_t|q_t = k)}{P(\mathbf{o}_t)} = \frac{P(q_t = k|\mathbf{o}_t)}{P(q_t = k)} \tag{2.36}$$

On the right-hand side, the MLP output is divided by $P(q_t = k)$, which is the prior probability of state k. The left-hand side of the equation is called the scaled likelihood, which replaces $b_k(\mathbf{o}_t)$ in the HMM framework. The denominator of the scaled likelihood does not affect classification since it is a constant for all the sub-word units.

There are several advantages of using the hybrid HMM/ANN approach over the HMM/GMM approach [10]. Some of them include:

- ANN provides discriminant-based training. It increases the probability of the correct sub-word unit while at the same time decreasing the probability of the competing sub-word units.
- It can generate any kinds of non-linear functions of the inputs.
- It does not make any assumptions on the statistical properties of the input features.
- Multiple streams of information can be combined using different ANNs trained with different sets of features.

2.4 Language modelling

The purpose of language modelling (LM) is to provide a model for describing the linguistic information between words. Mathematically, a language model can be formulated as the probability of a sequence of words $P(W_1^n|\Theta_l)$, which can be evaluated as follows.

$$\begin{aligned} P(W_1^n|\Theta_l) &= P(W_1, W_2, ..., W_n|\Theta_l) \\ &= \prod_{i=1}^{n} P(W_i|W_1, W_2, ..., W_{i-1}, \Theta_l) \end{aligned} \tag{2.37}$$

where W_1^n is a sequence of n words and Θ_l are the parameters of the language model. In (2.37), it suggests that the i^{th} word in the sequence depends on all the preceding words $\{W_1, W_2, ..., W_{i-1}\}$. It makes the estimation of the conditional probability $P(W_i|W_1, W_2, ..., W_{i-1}, \Theta_l)$ difficult since the sequence $\{W_1, W_2, ..., W_i\}$ may occur only a few times or even none in the training corpus. To deal with this issue, it is assumed that a word is dependent on only the previous $N - 1$ words. This approach is referred to as

N-gram language modelling. Under this assumption, (2.37) is rewritten as follows.

$$\begin{aligned} P(W_1^n|\Theta_l) &= \prod_{i=1}^{n} P(W_i|W_1, W_2, ..., W_{i-1}, \Theta_l) \\ &= \prod_{i=1}^{n} P(W_i|W_{i-N+1}, W_{i-N+2}, ..., W_{i-1}, \Theta_l) \end{aligned} \qquad (2.38)$$

where the entire word history is truncated to contain only the previous $N-1$ words. Common N-grams that are used in ASR are *trigram* ($N = 3$), *bigram* ($N = 2$) and *unigram* ($N = 1$).

Training of the N-gram probabilities can be done by simple frequency counts. For example, a trigram probability can be estimated by the following equation.

$$P(W_i|W_{i-2}, W_{i-1}) = \frac{C(W_{i-2}, W_{i-1}, W_i)}{C(W_{i-2}, W_{i-1})} \qquad (2.39)$$

where $C(W_{i-2}, W_{i-1}, W_i)$ is the number of times that the sequence $\{W_{i-2}, W_{i-1}, W_i\}$ appears in the training utterances and $C(W_{i-2}, W_{i-1})$ is the number of appearances of the sequence $\{W_{i-2}, W_{i-1}\}$.

Even when N is small, there could be some word sequences which are unseen in the training data. The N-gram probabilities of these sequences are zero due to zero occurrence. This is not useful since these word sequences could appear during recognition but they will never be considered as possible word transcriptions. One way to resolve this issue is to adopt a *back-off* language model [29]. In this method, if the occurrence of a certain N-gram exceeds a threshold, the estimation of its N-gram probability is the same as the usual approach as shown in (2.39). However, if the occurrence is below the threshold, its N-gram probability is *discounted*. The amount of reduction in the probability mass is re-distributed to those N-grams which have zero occurrence. Hence, an unseen N-gram will have some probability mass depending on its lower-order N-gram probability. For example, the probability of an unseen trigram depends on the bigram probability as shown below.

$$P(W_i|W_{i-2}, W_{i-1}) = B(W_{i-2}, W_{i-1})P(W_i|W_{i-1}) \qquad (2.40)$$

where $B(W_{i-2}, W_{i-1})$ is the back-off weight which ensures that the trigram probability is properly normalized.

2.5 Decoding

In Section 2.3 and Section 2.4, acoustic modelling and language modelling are discussed. Using Equations (2.6) and (2.7), the decoding problem as shown in Equation (2.2) can be written as follows.

$$\begin{aligned}
\hat{W} &= \underset{W_1^n}{\operatorname{argmax}}\{P(\mathbf{o}_1^T|W_1^n,\Theta_a)P(W_1^n|\Theta_l)\} \\
&= \underset{W_1^n}{\operatorname{argmax}}\left\{\left[\sum_{q_1^T} P(\mathbf{o}_1^T, q_1^T|W_1^n,\Theta_a)\right] P(W_1^n|\Theta_l)\right\} \\
&\approx \underset{W_1^n}{\operatorname{argmax}}\left\{\left[\max_{q_1^T} P(\mathbf{o}_1^T, q_1^T|W_1^n,\Theta_a)\right] P(W_1^n|\Theta_l)\right\} \qquad (2.41)
\end{aligned}$$

The above equation can be interpreted in the following way. The Viterbi algorithm finds the optimal HMM state sequence which represents the word sequence W_1^n. This maximum likelihood is scaled by the language model. Finally, the recognizer output is the best word sequence, which gives the greatest scaled maximum likelihood out of all the possible word sequences.

2.6 Search space representation

In Section 2.5, the mathematical formulation of the decoding problem is presented. In order to solve the decoding problem, the recognizer needs to first represent the acoustic model and the language model in a searchable network called the search space. Then, the recognizer performs a search algorithm on the search space for finding the best word sequence given the observation sequence, as expressed in (2.41).

In this section, two common approaches for representing the search space are briefly discussed. They are *re-entrant lexical tree* and *weighted finite state transducer* (WFST). The former approach is a *dynamic expansion* approach, whereas the latter approach is a *static expansion* approach [6].

2.6.1 Re-entrant lexical tree

In this approach, the lexicon is organized as a tree. The root node of the tree indicates the start of a word, while each of the leaf nodes marks the end of a word. Each arc represents a context-dependent sub-word unit (for example, a triphone), which is substituted by a sub-word HMM. Hence, the entire lexical tree is a network of HMM states. The leave nodes are connected back to the root node. It allows the decoder to recognize the next word. Language model probabilities (for example, N-gram probabilities) are incorporated before re-entering to the tree. Figure 2.3 illustrates an example of a re-entrant lexical tree.

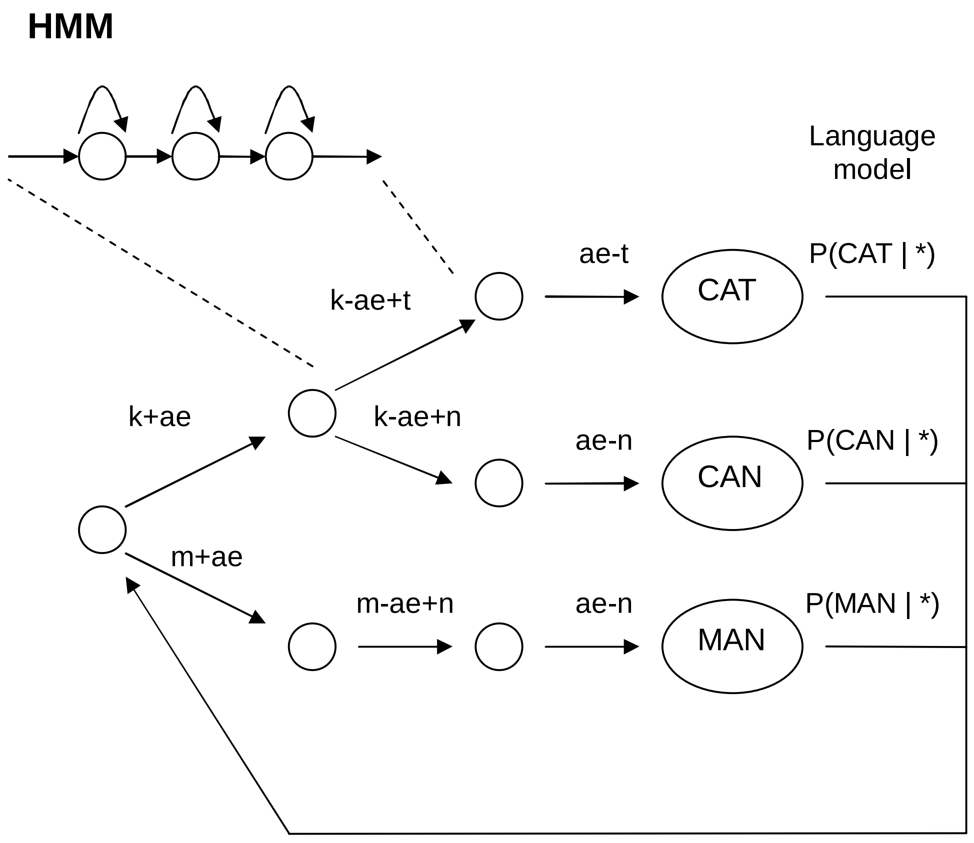

Figure 2.3: A re-entrant lexical tree. Language model probabilities are incorporated into the search at the leaf nodes of the tree. The asterisks in the langauge model probabilities denote the word histories. For example, if the language model is trigram, the asterisks represent word histories consisting of two words.

To search for the optimal state sequence (or the equivalent word sequence), a *token-passing* algorithm is often used [64]. A token (also known as a hypothesis) is an entity, which resides in an HMM state. A token stores a score and a path record. The score of a token is basically the product of $V_t(i)$ in (2.17) and $P(W_1^n|\Theta_l)$. It is the maximum probability, which is scaled by the language model, of an HMM state sequence ended in state i at time t. As shown in (2.41), the major operation in the decoding problem is the Viterbi algorithm, which tries to find the optimal HMM state sequence. In the token-passing paradigm, it is equivalent to passing a token from its residing state to the succeeding states. When two tokens meet at a state, only the best token survives and stays at the state. Other tokens are discarded. The path record of a token stores the state sequence, or equivalently the word sequence, that the token has gone through.

From the definition of $V_t(i)$ in (2.17), one point to note is that the Viterbi algorithm compares the state sequences which have the same word history W_1^n. In other words, tokens having different word histories are not compared when they meet at an HMM state. It suggests that there can be more than one token in an HMM state [1]. This is a *dynamic expansion* approach. Physically, there is only one lexical tree. However, since it is possible to have multiple tokens in an HMM state, there can be a number of *virtual* tree copies in the physical tree. Each virtual tree copy is distinguished by a distinct word history. The search space is dynamically changing according to the number of distinct word histories amongst all the active tokens. Strictly speaking, a word history starts from the very first word. In practice, this is infeasible and thus a word history is shortened to include just the previous $N-1$ words (N-gram). Tokens having the same N-gram histories are compared.

One of the issues in this lexical tree approach is that word identities are only known at the leaf nodes of the tree. For example, in Figure 2.3, suppose there is a token on the /k+ae/ arc. The identity of the next word is not known until it reaches either the "CAT" node or the "CAN" node. It implies that it is impossible to incorporate the language model probability on the /k+ae/ arc. Language model knowledge can only be employed at the leaf nodes (word-end nodes). It decreases the pruning efficiency of the recognizer (Beam pruning will be discussed in Section 2.7.1). First, many tokens remain active in the beginning of the tree because no langauge model knowledge is employed to help discard less promising tokens. Second, it introduces a drastic change of token scores at the leaf nodes, which could probably eliminate many potential tokens.

A solution to this problem is to factor the language model probabilities and apply them as early as possible. In the previous example, suppose the language model is trigram and the word history of the token is {MAN CAN}. Although the exact identity of the next word is not known on the /k+ae/ arc, there are only two possibilities (CAT or CAN) for the next word. Therefore, a portion of the probabilities can be factored and employed on this arc. Specifically, the maximum of $P(\text{CAT}|\text{MAN CAN})$ and $P(\text{CAN}|\text{MAN CAN})$ is found and incorporated to the token score. Later, suppose the token enters the /k−ae+t/ arc. In this case, the identity of the next word can be determined since there is only one possibility (CAT). As a portion has already been applied, it needs to only consider the residual probability, that is, the difference between the portion and $P(\text{CAT}|\text{MAN CAN})$. This technique is known as *langauge model lookahead* [58, 46]. It tries to smear the token scores along the path of the tree.

2.6.2 Weighted finite state transducer (WFST)

The use of weighted finite state transducers (WFST) [36, 35] in ASR is a *static expansion* approach, in which the entire search space is fully expanded. This is opposite to the re-entrant lexical tree approach. In that approach, the search space is dynamically changing according to the number of virtual tree copies in the physical tree. In the WFST approach, there are no virtual copies. The WFST by itself is the entire search space. When tokens meet at an HMM state, it is guaranteed that they have the same word history and thus can be compared.

Basically, a WFST is a finite state machine with a number of states and transitions. Figure 2.4 shows an example of a WFST. For simplicity, there are only two words in the vocabulary and the language model is bigram. In contrast with the re-entrant lexical tree illustrated in Figure 2.3, the entire search space is fully expanded. Each transition has an input symbol, an output symbol and an associated weight. Symbols can be real symbols or empty symbols (ϵ), which are merely placeholders. Here, the input symbols are triphone or biphone labels. The output symbols are words. The WFST weights are the language model probabilities. Each triphone or biphone label is modelled by an HMM. Hence, the entire WFST is essentially a network of HMM states. The transition probabilities within the HMM can also be seen as the weights of the search network.

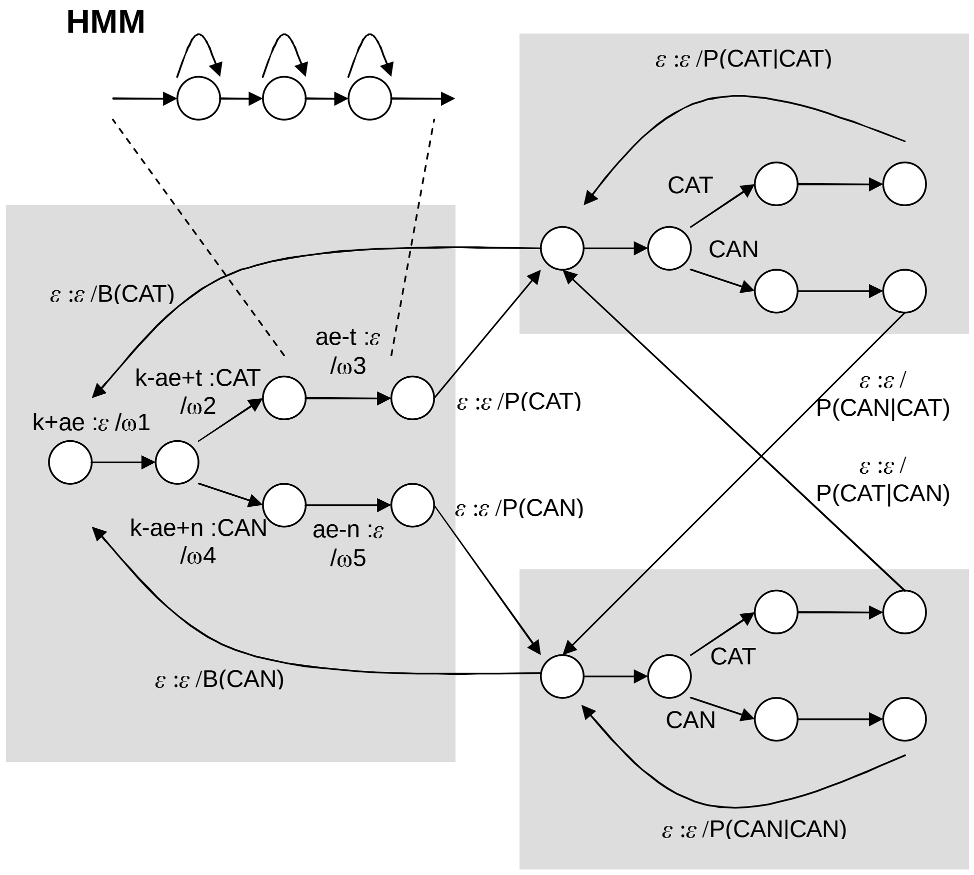

Figure 2.4: Search space represented by a WFST. There are only two words in the vocabulary. The language model is bigram. Each WFST transition $x : y/\omega$ has three attributes. x is an input symbol representing a triphone or biphone label. y is an output label representing a word. Labels can be ϵ, which are empty labels. ω is the weight representing the language model probability $P(.)$ or the back-off weight $B(.)$. Each triphone or biphone label is modelled by an HMM.

Like the lexical tree approach, token-passing can be performed during the search for the optimal word sequence. However, since the WFST approach is a static expansion approach, there is no need to compare the word histories when two tokens meet at an HMM state.

The first step for constructing the entire search space in one WFST is to represent each knowledge source by a constituent WFST. Then, these constituent WFSTs are combined into one integrated WFST using the composition algorithm [36, 35, 50]. The knowledge sources include sub-word context-dependency, lexical knowledge and language model. The fully expanded WFST can be very large, especially when the language model is complex. Therefore, optimization algorithms, for example, determinization, weight-pushing and minimization, are performed on the integrated WFST in order to reduce the number of WFST states and transitions to a minimum [36, 35]. More details on these algorithms will be presented in Chapter 3 where we introduce a WFST-based decoder called *Juicer*, which is the underlying recognition engine in our proposed embedded ASR system.

There are two main advantages in the WFST approach. First, the decoder design is simple because all the knowledge sources have been integrated into one compact WFST. The knowledge sources are *decoupled* from the Viterbi search and therefore the decoder does not need to perform any combination of knowledge sources during decoding. For example, the decoder does not need to check the N-gram histories of the tokens during the Viterbi search. The second advantage is that the fully integrated transducer can be further optimized by algorithms, such as, determinization, minimization and weight-pushing. These optimization algorithms remove redundancy, for example, redundant HMM states, in the search space. It decreases the number of tokens residing in the search space, which makes the recognizer run faster.

2.7 Search algorithm

With the search space represented in a network of HMM states, the speech recognizer can now perform a search algorithm for finding the optimal word sequence. There are two main types of search algorithms, namely *time-synchronous search* and *time-asynchronous search*.

2.7.1 Time-synchronous search

In time-synchronous search, the decoder iterates through all the tokens in the search space at each time step (speech frame). The tokens propagate through the search space according to the Viterbi algorithm. All the tokens finish their propagation before entering into the next time step. It is a *breadth-first* search.

In practice, it is infeasible if there is a token in every HMM state. The number of tokens that need to be propagated would be too large for the decoder. Therefore, *pruning* is essential for practical applications with the cost of introducing search errors. One of the common pruning techniques is called *beam pruning* [44, 43]. At each time step, the maximum score amongst all the tokens is found. Then, a *pruning threshold* is determined by subtracting a certain value called the *pruning beamwidth* from the maximum score. If the score of a token is above the pruning threshold, the token remains active. Otherwise, the token is discarded. This approach is called *time-synchronous Viterbi beam search*. Another common pruning technique is called *histogram pruning* [58]. In this technique, the number of active tokens is kept within a certain limit. At each time step, the tokens are sorted by their scores. The decoder only allows the top N tokens to propagate. Other tokens are deactivated.

Figure 2.5 describes the pseudocode of the time-synchronous Viterbi beam search algorithm.

Algorithm 1 Time-synchronous Viterbi beam search

```
1:  /* $\tilde{Q}_t$ is a set of HMM states having a token at time $t$ */
2:  $\tilde{Q}_1 \leftarrow Q_{word-start}$
3:  $score_{i,1} \leftarrow 0$ for all $i \in \tilde{Q}_1$
4:
5:  for $t = 1$ to $T$ do
6:      $\mathbf{o}_t \leftarrow Feature\_extraction(Frame_t)$
7:
8:      $max\_score \leftarrow \max(score_{i,t})$ for all $i \in \tilde{Q}_t$
9:      $pruning\_threshold \leftarrow max\_score - pruning\_beamwidth$
10:     $\tilde{Q}_{t+1} \leftarrow \{\}$
11:
12:     for all $i \in \tilde{Q}_t$ do
13:         if $score_{i,t} > pruning\_threshold$ then
14:             $log\_emis\_prob \leftarrow Emission\_prob\_calc(\mathbf{o}_t, i)$
15:             $\mathcal{V} \leftarrow Viterbi\_search(log\_emis\_prob, i, t)$
16:             $\tilde{Q}_{t+1} \leftarrow \tilde{Q}_{t+1} \cup \mathcal{V}$
17:         end if
18:     end for
19: end for
20:
21: $\mathcal{Q} \leftarrow \tilde{Q}_{T+1} \cap Q_{word-end}$
22: $best\_token \leftarrow \underset{i \in \mathcal{Q}}{\operatorname{argmax}}(score_{i,T+1})$
```

Figure 2.5: Pseudocode of time-synchronous Viterbi beam search

Line 2 and 3 are some initializations. In Line 2, a token is put to each HMM state at the start of each word. $Q_{word-start}$ is a set of word-starting HMM states. $\tilde{Q}_1$, which is basically the same as $Q_{word-start}$, is a set of HMM states that have a token at $t = 1$. The score of each token is reset in Line 3.

After the initialization, the algorithm begins to process each frame of speech. In Line 6, an acoustic feature vector, $\mathbf{o}_t$, is generated by feature extraction for each speech frame.

Then, the pruning threshold is set in Line 8 and 9.

After feature extraction and setting the pruning threshold, the algorithm iterates through all the HMM states that have a token (Line 12 - 18). If the token stays above the pruning threshold, the emission probability of that state is calculated (Line 14). For computational efficiency, emission probability is calculated in the logarithmic domain. Hence, multiplication of probabilities becomes addition of log probabilities. Once the emission probability is calculated, Viterbi search is performed on that particular HMM state (Line 15). Token propagation takes place during this process. It returns a set of new HMM states, $\mathcal{V}$, which contain new tokens after token-passing. The new tokens are accumulated in another set, $\tilde{Q}_{t+1}$, which is prepared for the next speech frame.

Once all the speech frames have been processed, the best token is found amongst all the word-end HMM states denoted by $\mathcal{Q}$ (Line 21 - 22). The best token records its path from which the word transcription can be determined.

Figure 2.6 shows the pseudocode of the $Viterbi\_search()$ function. The for-loop iterates through all the succeeding states of i (Line 2 - 9). For each succeeding state, new_score is calculated by summing the score at i, the log emission probability and the transition weight from i to i_suc. The transition weight can either be the HMM transition probability for within-HMM transitions or the WFST transition weight for cross-HMM transitions. If new_score is greater than the score at i_suc, the new_score will update the score at i_suc (Line 5). The path record of the original token at i_suc is replaced by the path record at i (Line 6). The function returns a set of HMM states, $\mathcal{V}$, which contain new tokens (Line 7 and 10).

```
Algorithm 2    $\mathcal{V} \leftarrow Viterbi\_search(log\_emis\_prob, i, t)$
 1: $\mathcal{V} \leftarrow \{\}$
 2: for all $i\_suc$ states that succeeds State $i$ do
 3:     $new\_score \leftarrow score_{i,t} + log\_emis\_prob + trans\_weight(i, i\_suc)$
 4:     if $new\_score > score_{i\_suc,t+1}$ then
 5:         $score_{i\_suc,t+1} \leftarrow new\_score$
 6:         $path_{i\_suc,t+1} \leftarrow path_{i,t}$
 7:         $\mathcal{V} \leftarrow \mathcal{V} \cup \{i\_suc\}$
 8:     end if
 9: end for
10: return $\mathcal{V}$
```

Figure 2.6: Pseudocode of the *Viterbi_search* function

2.7.2 Time-asynchronous search

In time-asynchronous search [27, 47], there is no need to propagate all the tokens before entering into the next time step. The decoder maintains a stack of tokens, which are sorted by the scores. The most promising token is pursued until the end of the speech. In other words, the tokens do not need to have the same time reference. It is a *depth-first* approach. The advantage of this approach is that the decoder does not require to propagate the unpromising tokens. However, the disadvantage is that it is difficult to decide which token to propagate, since the decision also relies on the part of the speech utterance that has not been decoded yet. Besides, pruning is non-trivial because the tokens have path records of different lengths. As a result, the scores of the tokens cannot be directly compared.

2.8 Performance metrics

ASR systems are evaluated mainly on two performance metrics - word accuracy and timing performance. The accuracy of an ASR system is measured by *word accuracy rate*, which is defined by the following equation [63].

$$\text{Word accuracy rate (\%)} = \frac{N - S - D - I}{N} \times 100\% \tag{2.42}$$

where N is the total number of words; S is the number of word substitutions; D is the number of word deletions and I is the number of word insertions. In some literature, *word error rate* is used instead of word accuracy rate.

$$\text{Word error rate (\%)} = 100\% - \text{Word accuracy rate (\%)} \tag{2.43}$$

The timing performance is measured by *real-time factor*. The following equation determines the real-time factor.

$$\text{Real-time factor} = \frac{\text{Decoding time}}{\text{Speech duration}} \tag{2.44}$$

It can be seen that an ASR system is a faster system when its real-time factor is smaller. A real-time factor of 1 suggests that the time required for recognition is equal to the duration of the input speech.

2.9 Summary

In this chapter, the decoding problem is described. The building blocks of a typical ASR system are presented. It explains different search space representation and search strategies. This chapter also mentions the performance metrics used for system evaluation.

3
WFST-based speech recognizer

As the first step towards developing an embedded ASR system, we start with a WFST-based speech recognizer called Juicer, which has run successfully on desktop platforms [41, 42]. In this chapter, the theory of WFST is briefly described. It is then followed by an overview of Juicer.

3.1 WFST theory

A weighted finite state transducer (WFST) is a finite state automaton consisting of states and transitions [36, 35]. Each transition has an input label, an output label and an associated weight. Figure 3.1 illustrates an example of a WFST.

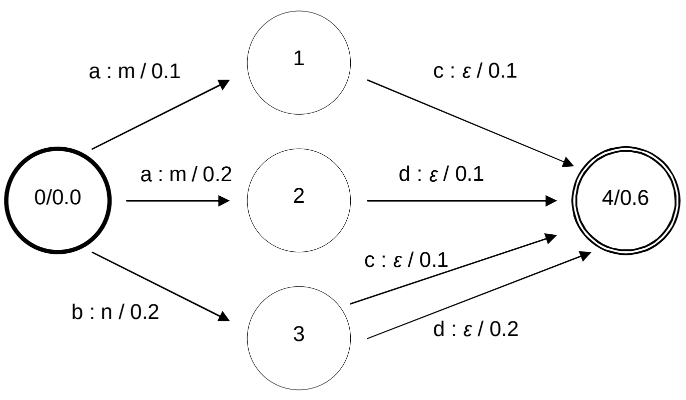

Figure 3.1: An example of a WFST

Each transition is denoted by $x : y/\omega$, where x is the input symbol, y is the output symbol and ω is the transition weight. A symbol can be an ϵ symbol, which represents an *empty* symbol. The bold circle is the *initial* state of the WFST, whereas the double-lined circle is the *final* state. Both states have an associated weight. The following paragraph gives the mathematical definition of a WFST [36].

Definition of WFST A weighted finite state transducer, T, over a semiring $\mathbb{K}$ is an 8-tuple $T = (\Sigma, \Delta, Q, I, F, E, \lambda, \rho)$, where

- Σ is the *input alphabet*
- Δ is the *output alphabet*
- Q is a set of *states*
- I is a set of *initial* states: $I \subseteq Q$
- F is a set of *final* states: $F \subseteq Q$
- E is a set of *transitions*: $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\}) \times \mathbb{K} \times Q$
- λ is the *initial weight function*: $\lambda : I \to \mathbb{K}$
- ρ is the *final weight function*: $\rho : F \to \mathbb{K}$.

Referring to Figure 3.1, the input alphabet consists of four symbols $\{a, b, c, d\}$ and the output alphabet has two symbols $\{m, n\}$. There are five states and seven transitions in the WFST. State 0 is the initial state with a weight of 0.0 and State 4 is the final state with a weight of 0.6.

In the definition of WFST, a term called *semiring* is mentioned. A semiring is a set $\mathbb{K}$ equipped with two binary operations ($\oplus$ and $\otimes$) and two identity elements ($\bar{0}$ and $\bar{1}$). A semiring satisfies the following four criteria [36].

- $(\mathbb{K}, \oplus, \bar{0})$ is a commutative *monoid*: For all $k \in \mathbb{K}$, $k \oplus \bar{0} = \bar{0} \oplus k = k$.
- $(\mathbb{K}, \otimes, \bar{1})$ is a *monoid*: For all $k \in \mathbb{K}$, $k \otimes \bar{1} = \bar{1} \otimes k = k$.
- $\otimes$ distributes over $\oplus$.
- $\bar{0}$ is an annihilator for $\otimes$: For all $k \in \mathbb{K}$, $k \otimes \bar{0} = \bar{0} \otimes k = \bar{0}$.

The semiring defines a set of mathematical operations on the *weights* of the WFST. Suppose there is a *path*, π, which starts from an initial state, i, and ends at a final state, f, of the WFST. The path maps an input symbol sequence $x \in \Sigma^*$ to an output symbol sequence $y \in \Delta^*$, where the asterisks indicate that there are at least one symbols in the sequences. Let us further assume that the path consists of N transitions denoted by

$(e_1, e_2, ..., e_N)$. The *path weight*, $w(\pi)$, is the $\otimes$-product of the initial state weight, the weights of the constituent transitions and the final state weight.

$$w(\pi) = \lambda(i) \otimes w(e_1) \otimes w(e_2) \otimes ... \otimes w(e_N) \otimes \rho(f) \tag{3.1}$$

If there is more than one path that maps the sequence x to y, the weight of this *sequence mapping*, $[T](x, y)$, is the $\oplus$-sum of all the associated paths.

$$[T](x, y) = \bigoplus_{\pi \in P(I,x,y,F)} w(\pi) \tag{3.2}$$

where $P(I, x, y, F)$ is a set of paths, which map x to y starting from an initial state to a final state of the WFST.

Depending on the type of the semiring, the functionality of a WFST is different. Table 3.1 lists some common semirings used in WFST. For speech recognition, *tropical* semiring is often used. The weights of the transitions represent negative log probabilities (weight $= -\log(\text{probability})$). The $\otimes$ operation corresponds to addition of the negative log probabilities. The $\oplus$ operation finds the minimum negative log probability (that is, the maximum positive log probability), which is essentially the *Viterbi* operation.

Semiring	$\mathbb{K}$	$\oplus$	$\otimes$	$\bar{0}$	$\bar{1}$
Boolean	$\{0, 1\}$	$\vee$	$\wedge$	0	1
Probability	$\mathbb{R}_+$	$+$	$\times$	0	1
Log	$\mathbb{R} \cup \{-\infty, +\infty\}$	$\oplus_{\log}$	$+$	$+\infty$	0
Tropical	$\mathbb{R} \cup \{-\infty, +\infty\}$	min	$+$	$+\infty$	0

Table 3.1: Common semirings used in WFST. $\vee$ and $\wedge$ are *logical-or* and *logical-and* operators respectively. $\mathbb{R}$ denotes the real number set. $\oplus_{\log}$ is defined as $x \oplus_{\log} y = -\log(\exp(-x) + \exp(-y))$.

One of the reasons why WFSTs are widely used in many applications is because there exists a set of WFST algorithms that can help represent different kinds of knowledge sources in a compact and optimal manner. Here are some of the WFST algorithms.

Composition Composition involves combination of two WFSTs into one integrated WFST [36, 35, 50]. Suppose T_1 is the first transducer mapping a sequence x to y and T_2 is the second transducer mapping the sequence y to z. The integrated WFST after composition, denoted by $T_1 \circ T_2$, maps the sequence x (which is the input sequence of T_1) to z (which is the output sequence of T_2). The weight of the mapping is the $\otimes$-product of the weights of the two constituent mappings. If there are more than one intermediate sequences y in the mapping of x to z, the weight is the $\oplus$-sum of all the intermediate mappings.

$$[T_1 \circ T_2](x, z) = \bigoplus_y [T_1](x, y) \otimes [T_2](y, z) \tag{3.3}$$

Figure 3.2 illustrates a graphical example of WFST composition, $T_1 \circ T_2$, over the tropical semiring. The states of $T_1 \circ T_2$ are labelled with a (S_1, S_2) pair, where S_1 and S_2 are the state indices of T_1 and T_2 respectively. As shown in the figure, it can be seen that the output symbols of T_1 are matched with the input symbols of T_2. The weights are semiring-multiplied ($\otimes$). For example, the transition $a : e/0.2$ of T_1 is combined with $e : m/0.2$ of T_2 since e is the common symbol. The weight after composition is 0.4 because $0.2 \otimes 0.2 = 0.4$ for the tropical semiring.

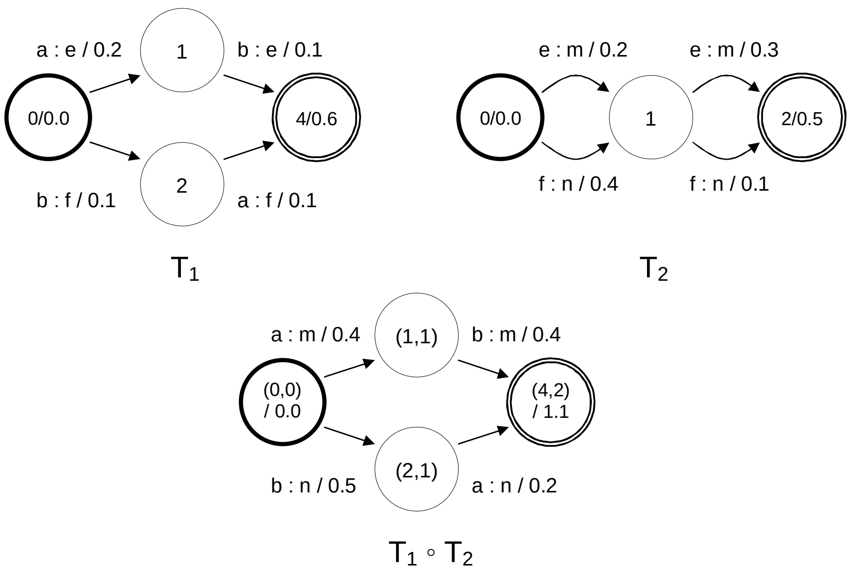

Figure 3.2: Composition of T_1 and T_2 over the tropical semiring.

Determinization A WFST is deterministic if there is only one unique initial state and no two transitions leaving any state have the same input label [36, 35]. Determinization, denoted by $det(T)$, is the algorithm which makes T deterministic. Figure 3.3 illustrates the determinization process. As shown in the figure, the transducer T is non-deterministic since there are two transitions, which share the same input label a, leaving the initial state. After determinization, the two transitions are replaced by one transition. The new weight is the $\oplus$-sum of the two weights. If the original weight is different from the new weight, the difference will be "pushed" backwards towards subsequent transitions. For example, the $d : \epsilon/0.1$ transition in T becomes $d : \epsilon/0.2$ since there is a remaining weight of 0.1 from $a : m/0.2$ after determinization. The resultant transducer is deterministic. Determinization removes redundancy in a non-deterministic transducer.

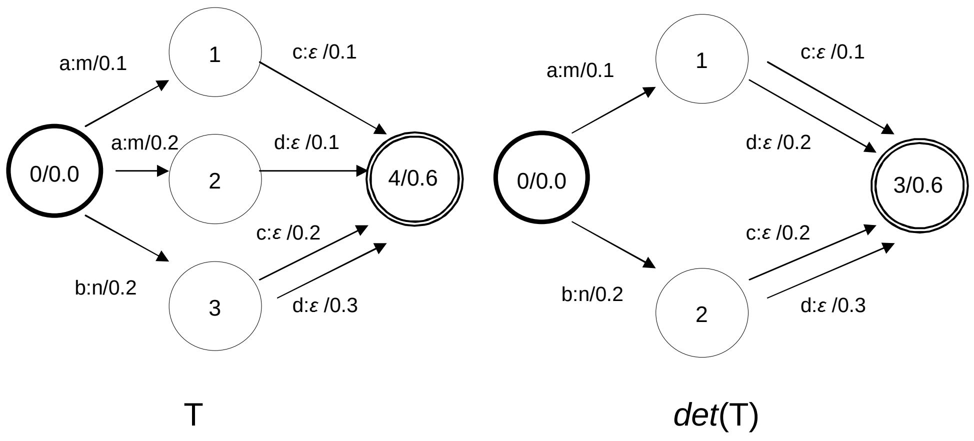

Figure 3.3: Determinization of T over the tropical semiring

Weight-pushing Weight-pushing refers to the algorithm which "pushes" the transition weights towards the initial state as much as possible [39]. There are two reasons for performing weight-pushing. The first reason is to allow the weights to be incorporated as early as possible, which helps improve the pruning efficiency. This is similar to the idea of language model lookahead as discussed in Section 2.6.1. The second reason is that weight-pushing is an essential step before performing another WFST optimization algorithm called minimization. Figure 3.4 shows the process of weight-pushing over the tropical semiring. The weights of T are pushed towards the initial state. One point to note is that weight-pushing does not change the weight of any path. It only re-distributes the weights along a path.

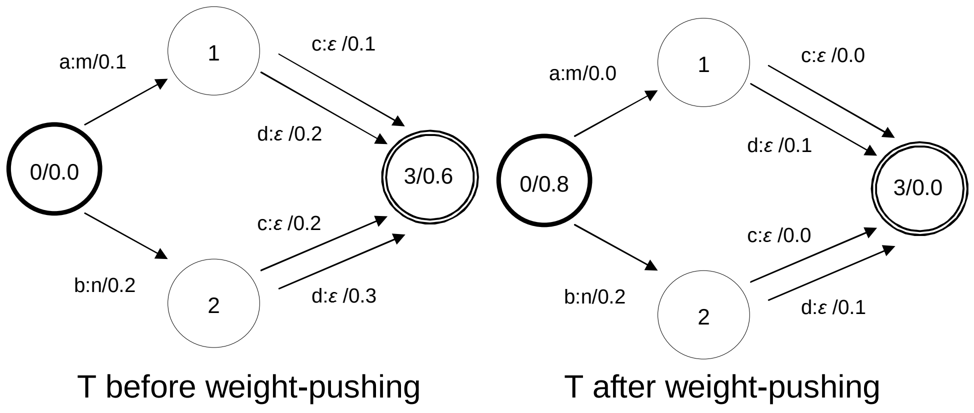

Figure 3.4: Weight-pushing of T over the tropical semiring

Minimization Minimization, denoted by $min(T)$, is an algorithm which reduces the number of states and transitions of a WFST to a minimum [36, 35]. In order to successfully perform minimization, weight-pushing is often needed to be carried out first. After weight-pushing, any sub-paths with the same input/output symbol sequences and weights are combined. Figure 3.5 shows the process of minimization.

On the left-hand side, the transducer is a deterministic WFST after weight-pushing. It can be seen that the last four transitions pointing towards the final state can actually be combined into two pairs of transitions. Minimization further removes redundancy in a WFST.

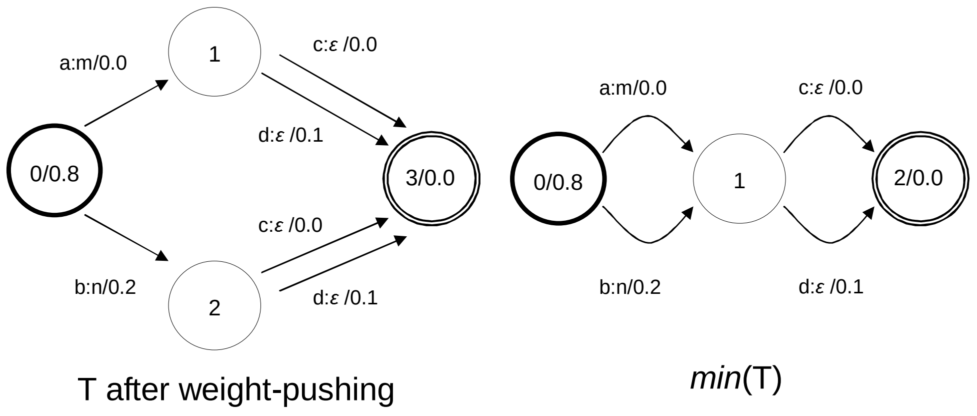

Figure 3.5: Minimization of WFST

3.2 Static WFST composition

In ASR applications, each knowledge source is represented by a WFST. Table 3.2 shows a list of knowledge sources typically used in an ASR system. Each WFST shown in the table maps an input symbol sequence to an output symbol sequence with an associated weight. For example, the language model WFST maps a sequence of words to the same sequence of words with the N-gram probability as the weight of the mapping.

Knowledge source	Input symbol	Output symbol	Weight
Language model / grammar (G)	Word	Word	N-gram probability
Lexicon (L)	Context-independent sub-word unit	Word	Pronunciation probability
Context-dependency (C)	Context-dependent sub-word unit	Context-independent sub-word unit	-
HMM topology (H)	HMM state emission distribution	Context-dependent sub-word unit	HMM transition probability

Table 3.2: Knowledge sources in a typical ASR system. Negative log probabilities are used as weights if the semiring is log or tropical.

These knowledge sources are integrated into one WFST by the composition algorithm. The integrated transducer is further optimized by determinization, weight-pushing and minimization to generate a compact search space. As discussed in Section 2.6.2, this approach is a *static* approach since the search space is fully expanded offline. There is no need to further expand the search space during decoding. Static integration of the WFSTs can be expressed as follows [38].

$$T = \pi_{\epsilon}(min(det(\tilde{H} \circ det(\tilde{C} \circ det(\tilde{L} \circ G))))) \tag{3.4}$$

In the equation, the $\tilde{\cdot}$ symbol means that the WFST is augmented with *auxiliary* symbols which are necessary for the success of determinization. Generally, if the L transducer contains *homophones*, $L \circ G$ is not determinizable. Homophones are words with the same sequence of context-independent (CI) sub-word units. For example, both “red” and “read” have the same CI sequence /r eh d/. In order to distinguish them, a distinct auxiliary symbol is added to each of the homophones. In this example, the CI sequences of “red” and “read” become /r eh d $\#_0$/ and /r eh d $\#_1$/ respectively. The other two WFSTs, C and H, are also modified to accept the newly-introduced auxiliary symbols. After composition and optimization, the π_ϵ operation replaces the auxiliary symbols by ϵ symbols. The final transducer T is a fully integrated transducer, which maps HMM state sequences to word sequences.

In some decoder implementation, the H transducer is not composed with the other WFSTs. Instead, only C, L and G are composed [40, 19].

$$T = \pi_{\epsilon}(min(det(\tilde{C} \circ det(\tilde{L} \circ G)))) \tag{3.5}$$

where T maps a sequence of context-dependent (CD) sub-word units to a sequence of words. During decoding, a CD symbol is substituted by its corresponding HMM, as previously shown in Figure 2.4.

3.3 Overview of Juicer

Juicer is a software package consisting of two main parts [41, 42]. The first part is for WFST generation. Specifically, the software tools provided by the Juicer package generate three WFSTs (C, L and G) from various knowledge sources, which include HMM definition (for generating C), pronunciation dictionary (for generating L) and language model (for generating G). Based on Equation (3.5), the three constituent WFSTs are composed into one fully-integrated WFST, which is further optimized by determinization, weight-pushing and minimization. For WFST composition and optimization, Juicer relies on third-party tools such as AT&T FSM Library [37] and MIT FST Toolkit [24]. The second part of the software package is a WFST-based time-synchronous speech recognizer. At the initialization stage, the fully-integrated WFST search space and the HMM definition are loaded into the recognizer. During decoding, acoustic features are inputted to the recognizer.

In the original implementation of Juicer, the recognizer decodes speech utterances based on acoustic features that are obtained from third-party tools, such as Hidden Markov Model Toolkit (HTK) [63]. In order to develop a complete ASR system, we have implemented a feature extraction module that works with the Juicer recognizer. The feature extraction module converts audio speech samples to a sequence of mel frequency cepstral coefficients (MFCC).

3.4 Summary

In this chapter, we introduce a WFST-based speech recognizer called Juicer, which has run successfully in many ASR experiments on the desktop environment. The theory of WFST is discussed. We also briefly describe how Juicer generates the WFST search space for decoding. In the following chapters, we will develop an embedded ASR system based on Juicer. We will investigate any enhancements required for building an ASR system on the target embedded platform.

4 Fixed-point speech recognition system

For desktop applications, speech recognition algorithm often uses floating-point arithmetic. However, in many embedded systems, a hardware floating-point processing unit is absent. Hence, it is necessary to consider a fixed-point implementation of the speech recognizer and evaluate its performance in terms of recognition accuracy and decoding speed.

A typical speech recognition system involves processing numerical data with different dynamic ranges. In order to minimize the quantization error, it is essential to assign different precision formats to different data types. In this chapter, we propose a framework for converting data formats from floating-point to fixed-point. The speech recognition algorithm is partitioned into three sub-tasks, namely, *feature extraction*, *emission probability calculation* and *Viterbi search*. The algorithm of each sub-task is presented, followed by a framework for data format conversion.

4.1 Feature extraction

4.1.1 Algorithm

The purpose of feature extraction is to transform a speech signal into a sequence of acoustic feature vectors. One of the commonly used feature extraction algorithms is mel frequency cepstral coefficients (MFCC) [17]. Figure 4.1 shows the flow diagram of the

MFCC algorithm.

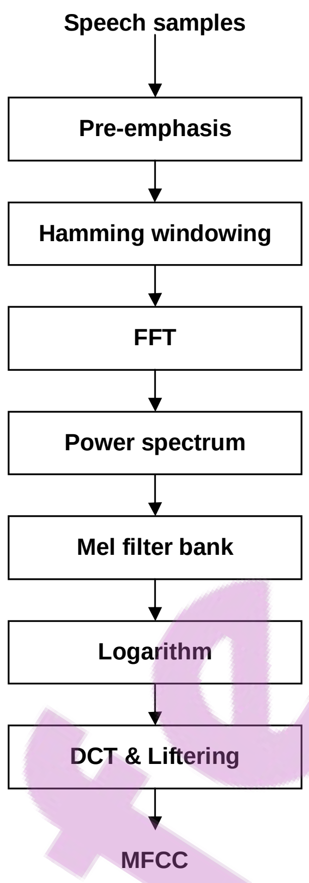

Figure 4.1: Flow diagram of MFCC

The individual parts of the MFCC algorithm are described as follows.

Pre-emphasis Raw speech samples are filtered by a pre-emphasis filter, which is a high-pass FIR filter. In (4.1), $\breve{s}_{t+1}$ and s_t are the filtered speech sample and the raw speech sample at time $t+1$ and t respectively. α is the filter coefficient. Typical value of α is 0.97.

$$\breve{s}_{t+1} = 1.0 - \alpha s_t \tag{4.1}$$

Hamming windowing A speech frame of N samples is extracted from the filtered signal. The speech frame is multiplied by a Hamming window function. Equation (4.2) shows the process of Hamming windowing at time t. In the equations, $\tilde{s}_{t+n}$ is the Hamming-windowed speech sample and h_n is the Hamming window weighting coefficient. The duration of the Hamming window is the same as the speech frame.

$$\begin{aligned} \tilde{s}_{t+n} &= h_n \breve{s}_{t+n} && 0 \le n < N \\ h_n &= 0.54 - 0.46 \cos\left(\frac{2\pi n}{N}\right) && 0 \le n < N \end{aligned} \tag{4.2}$$

FFT The frequency spectrum, S_k, is found by performing FFT on the windowed speech frame, $\tilde{s}_{t+n}$, where $0 \leq k < K$ and K is the number of FFT points. The frame is zero-padded if $N < K$. The basic operations of FFT are multiplication by twiddle factors, $W_{twiddle}$, and summation of intermediate results [51].

Power spectrum The frequency spectrum is squared to obtain the power spectrum, Y_k, for $0 \leq k < \frac{K}{2}$.

$$Y_k = |S_k|^2 \quad 0 \leq k < \frac{K}{2} \tag{4.3}$$

Mel filter bank The power spectrum is filtered by a series of triangular filters located logarithmically along the frequency axis. The output of the m^{th} mel filter, X_m, is expressed in (4.4), where $F_{m,k}$ is the frequency response of the m^{th} filter at frequency index k. The total number of mel filters in the filter bank is M. Typical value of M is 26.

$$X_m = \sum_{k=0}^{\frac{K}{2}-1} F_{m,k} Y_k \quad 0 \leq m < M \tag{4.4}$$

Logarithm The logarithms of the mel filter outputs, $\log X_m$ for $0 \leq m < M$, are taken.

DCT and Liftering DCT is performed on the log mel filter outputs, which are subsequently weighted by the liftering coefficients. The final results are D_{static} MFCC coefficients. Typically, 13 coefficients ($D_{static} = 13$) are calculated. Equations (4.5) to (4.7) demonstrate the DCT and liftering operations. In the equations, $o^{(d)}$ is the d^{th} MFCC coefficient, l_d is the liftering coefficient and L is a constant with a typical value of 22. The weighting coefficients, $w_{m,d}$, combines the liftering coefficient with the DCT coefficient.

$$o^{(d)} = \sum_{m=0}^{M-1} \log(X_m) w_{m,d} \qquad 0 \leq d < D_{static} \tag{4.5}$$

$$w_{m,d} = l_d \sqrt{\frac{2}{M}} \cos\left[\frac{\pi d}{M}(m + 0.5)\right] \quad 0 \leq d < D_{static}; 0 \leq m < M \tag{4.6}$$

$$l_d = 1 + \frac{L}{2} \sin\left(\frac{\pi d}{L}\right) \qquad 0 \leq d < D_{static} \tag{4.7}$$

The MFCC coefficients calculated from the above procedure are known as static coefficients. Temporal correlation between static features is modelled by first-order and second-order dynamic features. As discussed in Section 2.2, delta and delta-delta coefficients are concatenated with the static coefficients to constitute a feature vector, which is denoted by the following expression.

$$\mathbf{o}_t = \left[o_t^{(0)}, o_t^{(1)}, ..., o_t^{(D-1)}\right] \tag{4.8}$$

where D is the dimension of the feature vector. Typically, a feature vector consists of 39 coefficients ($D = 39$). Amongst the 39 coefficients, there are 13 static, 13 delta and 13 delta-delta coefficients. The t subscript in (4.8) indicates the time index at which the feature vector is calculated. The final stage of feature extraction is cepstral mean variance normalization (CMVN), where the mean and the variance of each coefficient are normalized to 0 and 1 respectively.

4.1.2 Feature extraction with fixed-point formats

Table 4.1 shows the data types and their fixed-point formats within the feature extraction sub-task.

Stage	Data type	Notation	Fixed-point format
Pre-emphasis	Raw speech	s_t	Q15.0
	Pre-emphasis filter coefficient	α	Q0.15
	Pre-emphasized speech	$\breve{s}_t$	Q16.15
Hamming windowing	Hamming window function	h_n	Q1.14
	Hamming-windowed speech	$\tilde{s}_t$	Q(31-f).f
FFT	Twiddle factor	$W_{twiddle}$	Q1.14
	Frequency spectrum	S_k	Q(31-f).f
Power spectrum	Power spectrum	Y_k	Q(63-2f).2f
Mel filter bank	Mel filter frequency response	$F_{m,k}$	Q1.f
	Mel filter output	X_m	Q(63-2f+5).(2f-5)
Logarithm	Logarithm of mel filter outputs	$\log X_m$	Q5.10
DCT and Liftering	Combined DCT and liftering coeff.	$w_{m,d}$	Q2.13
	MFCC features	$o_t^{(d)}$	Q11.20

Table 4.1: Fixed-point formats of various data types in the MFCC algorithm

The following paragraphs describe how the fixed-point formats are derived.

Pre-emphasis The raw speech samples in this study are 16-bit signed integers. Thus, their formats are Q15.0. The filter coefficient, α, is 0.97. We express α in 16-bit Q0.15 format. The raw speech sample is multiplied by α. Therefore, the filtered speech samples are in 32-bit Q16.15 formats.

Hamming windowing The range of the Hamming window function, h_n, is between 0.08 and 1.00 inclusive. Hence, its format is Q1.14. According to (4.2), the pre-emphasized speech samples are multiplied by the Hamming window function. As a result, the Hamming-windowed speech samples are in Q18.29 formats. To reduce the number of bits involved in subsequent computation, these speech samples are truncated to 32 bits. Experiments are performed to decide how many bits can be truncated in order to demonstrate the least degradation in recognition accuracy. The results are shown in Section 4.4. In the general form, the Hamming-windowed speech samples are in Q(31-f).f formats, where f is the number of fractional bits after truncation.

FFT The real and imaginary parts of the twiddle factors are between -1 and 1 inclusive and thus Q1.14 formats are used. Multiplication by the twiddle factors increases the number of fractional bits by 14. In order to keep the frequency spectrum, S_k, in 32-bit Q(31-f).f format, the least significant 14 bits are truncated from the intermediate product after each multiplication operation.

Power spectrum Squaring the frequency spectrum doubles the number of fractional bits. The power spectrum is therefore in 64-bit Q(63-2f).2f format.

Mel filter bank As shown in (4.4), the power spectrum is multiplied by the frequency response of the mel filter. To keep the product, $F_{m,k}Y_k$, in 64-bit, the least significant f bits of Y_k are first truncated before multiplication. Since $F_{m,k}$ are in Q1.f formats, the formats of $F_{m,k}Y_k$ are the same as those of Y_k. The summation in (4.4) can increase a maximum of 5 integer bits (see Appendix A). Hence, before summation, the least significant 5 bits of $F_{m,k}Y_k$ are truncated to leave enough bits in order to avoid possible overflows. The final mel filter outputs, X_m, are in 64-bit Q(63-2f+5).(2f-5) formats.

Logarithm The logarithm of X_m, $\log(X_m)$, is represented by a 16-bit Q5.10 number, which is wide enough to cover the range of X_m after the logarithm is taken.

DCT and Liftering The range of $w_{m,d}$ can be determined by considering the ranges of l_d, $\sqrt{2/M}$ and cos(.), which are about (2.57, 12.00), 0.28 and (-1, 1) respectively. Combining all these together, the range of $w_{m,d}$ is (-3.36, 3.36), which can be represented by a 16-bit Q2.13 format. The format of the product, $\log(X_m)w_{m,d}$, is therefore Q8.23. Summation in Equation (4.5) can increase at most 3 more integer bits (see Appendix B). To avoid possible overflows, the least significant 3 fractional bits of the intermediate products, $\log(X_m)w_{m,d}$, are truncated before summation. The formats of the MFCC features, $o^{(d)}$, have 3 fewer fractional bits and thus in 32-bit Q11.20 formats.

The final two stages of feature extraction are finding the dynamic features and CMVN. The precision formats of the delta and delta-delta coefficients after CMVN are kept the same as the static coefficients. All the coefficients in a feature vector are in 32-bit Q11.20 formats.

4.2 Emission probability calculation

4.2.1 Algorithm

In our implementation, Gaussian mixture models (GMM) are used for modelling the emission probability distributions. In Section 2.3.5, HMM/GMM ASR systems have already been discussed. In this section, the details on implementation are presented. Given an observation feature vector $\mathbf{o}_t$, the emission probability function in an HMM state j is modelled by a sum of weighted Gaussian mixtures.

$$\begin{aligned} b_j(\mathbf{o}_t) &= \sum_{m=1}^{M} b_{jm}(\mathbf{o}_t) \\ &= \sum_{m=1}^{M} c_{jm}\mathcal{N}(\mathbf{o}_t, \boldsymbol{\mu}_{jm}, \boldsymbol{\Sigma}_{jm}) \\ &= \sum_{m=1}^{M} c_{jm}\frac{1}{(2\pi)^{\frac{D}{2}}|\boldsymbol{\Sigma}_{jm}|^{\frac{1}{2}}}\exp\left[-\frac{1}{2}\left(\mathbf{o}_t-\boldsymbol{\mu}_{jm}\right)^T\boldsymbol{\Sigma}_{jm}^{-1}\left(\mathbf{o}_t-\boldsymbol{\mu}_{jm}\right)\right] \end{aligned} \tag{4.9}$$

where $b_{jm}(\mathbf{o}_t)$ is the probability density function of the weighted m^{th} Gaussian mixture. The mean vector and the co-variance matrix of the Gaussian mixture are denoted by $\boldsymbol{\mu}_{jm}$ and $\boldsymbol{\Sigma}_{jm}$ respectively. Since the coefficients of a feature vector are assumed to be independent, $\boldsymbol{\Sigma}_{jm}$ is a diagonal matrix. The total number of Gaussian mixtures is M per HMM state. The weight of the m^{th} Gaussian mixture is c_{jm}.

In practice, it is more convenient to express $b_{jm}(\mathbf{o}_t)$ and $b_j(\mathbf{o}_t)$ in their logarithmic forms since multiplication of probabilities becomes addition in the logarithmic domain. Due to the assumption that the co-variance matrix is diagonal, the logarithm of a weighted Gaussian mixture can be expressed by the following equation.

$$\log b_{jm}(\mathbf{o}_t) = C_{jm} + g_{jm} + \sum_{d=0}^{D-1}\left(o_t^{(d)} - \mu_{jm}^{(d)}\right)^2 v_{jm}^{(d)} \tag{4.10}$$

where $\mu_{jm}^{(d)}$ is the d^{th} dimension of the $\boldsymbol{\mu}_{jm}$ mean vector. C_{jm}, $v_{jm}^{(d)}$ and $g_{jm}^{(d)}$ are constants defined as follows.

$$C_{jm} = \log c_{jm} \tag{4.11}$$

$$v_{jm}^{(d)} = \frac{-1}{2\left(\sigma_{jm}^{(d)}\right)^2} \tag{4.12}$$

$$g_{jm} = -\frac{1}{2}\left(D\log(2\pi) + \sum_{d=0}^{D-1}\log\left(\sigma_{jm}^{(d)}\right)^2\right) \tag{4.13}$$

The symbol $\left(\sigma_{jm}^{(d)}\right)^2$ is the d^{th} feature variance, which is the d^{th} diagonal element of the co-variance matrix.

The log emission probability, $\log b_j(\mathbf{o}_t)$, can be evaluated recursively by the following equation.

$$\log b_j(\mathbf{o}_t) = ((\log b_{j1}(\mathbf{o}_t) \oplus \log b_{j2}(\mathbf{o}_t)) \oplus \log b_{j3}(\mathbf{o}_t)) \oplus ... \oplus \log b_{jM}(\mathbf{o}_t) \tag{4.14}$$

where the $\oplus$ symbol represents the *log-add* operator having the following definition.

$$x \oplus y = \log(\exp(x) + \exp(y)) \tag{4.15}$$

4.2.2 Emission probability with fixed-point formats

Table 4.2 shows the various data types and their fixed-point formats during the calculation of the emission probabilities.

As discussed in Section 4.1.2, the MFCC features are in 32-bit Q11.20 formats. In order to reduce the number of bits in subsequent calculations, each feature coefficient is truncated to 16 bits. Since the dynamic range of each coefficient is different, a separate quantizer is built for each coefficient [9]. As shown in (4.10), the Gaussian mixture mean is subtracted from the MFCC features. Therefore, they have the same fixed-point format. The dynamic range of each mean component, denoted by $R_{\mu^{(d)}}$, is expressed as follows.

$$R_{\mu^{(d)}} = \left(\min_{j\in Q, 1\le m\le M}\mu_{jm}^{(d)}, \max_{j\in Q, 1\le m\le M}\mu_{jm}^{(d)}\right) \quad 0 \le d < D \tag{4.16}$$

Each Gaussian mixture in the entire set of HMM states, Q, is examined for determining the dynamic ranges of the mean components. A 16-bit fixed-point format is found for each $R_{\mu^{(d)}}$. The fixed-point formats should have a maximum number of fractional bits. Additionally, they should be able to accommodate their corresponding $R_{\mu^{(d)}}$ dynamic ranges. The fixed-point formats are represented by Q(15-e_d).e_d, where e_d is the number of fractional bits for the d^{th} mean component.

Similarly, the dynamic ranges of $v_{jm}^{(d)}$ are found for each dimension. The fixed-point formats are represented by Q(15-i_d).i_d, where i_d is the number of fractional bits for the

Data type	Notation	Fixed-point format
MFCC features (before truncation)	$o_t^{(d)}$	Q11.20
MFCC features (after truncation)	$o_t^{(d)}$	Q(15-e_d).e_d
Mixture mean	$\mu_{jm}^{(d)}$	Q(15-e_d).e_d
Constant	$v_{jm}^{(d)}$	Q(15-i_d).i_d
Distance from mean	$o_t^{(d)} - \mu_{jm}^{(d)}$	Q(15-e_d).e_d
Squared distance from mean (before truncation)	$\left(o_t^{(d)} - \mu_{jm}^{(d)}\right)^2$	Q(31-2e_d).2e_d
Squared distance from mean (after truncation)	$\left(o_t^{(d)} - \mu_{jm}^{(d)}\right)^2$	Q(31-2e_d).($2e_d - 16$)
Scaled squared distance (before truncation)	$\left(o_t^{(d)} - \mu_{jm}^{(d)}\right)^2 v_{jm}^{(d)}$	Q(47-2$e_d - i_d$).($2e_d + i_d - 16$)
Scaled squared distance (after truncation)	$\left(o_t^{(d)} - \mu_{jm}^{(d)}\right)^2 v_{jm}^{(d)}$	Q(15-p).p
Log Gaussian mixture weight	C_{jm}	Q(15-p).p
Constant	g_{jm}	Q(15-p).p
Log weighted Gaussian mixture	$\log b_{jm}(\mathbf{o}_t)$	Q(15-p).p
Log emission probability	$\log b_j(\mathbf{o}_t)$	Q(15-p).p

Table 4.2: Fixed-point formats of various data types in the GMM emission probability calculation

d^{th} component.

$$R_{v^{(d)}} = \left(\min_{j\in Q,1\leq m\leq M} v_{jm}^{(d)}, \max_{j\in Q,1\leq m\leq M} v_{jm}^{(d)}\right) \quad 0 \leq d < D \tag{4.17}$$

Table 4.3 shows the fixed-point formats of $\mu_{jm}^{(d)}$ (also applied to $o_t^{(d)}$) and $v_{jm}^{(d)}$ for each of the 39 dimensions. These formats are determined by examining all the HMM states trained using the training data set of the Resource Management (RM1) corpus [52].

Since the fixed-point formats of $o_t^{(d)}$ and $\mu_{jm}^{(d)}$ are the same, they share the same formats as $\left(o_t^{(d)} - \mu_{jm}^{(d)}\right)$. Squaring $\left(o_t^{(d)} - \mu_{jm}^{(d)}\right)$ doubles the number of fractional bits. To keep it in 16-bit, the least significant 16 bits are truncated. After that, multiplication by $v_{jm}^{(d)}$ increases the number of the fractional bits by i_d. Each $\left(o_t^{(d)} - \mu_{jm}^{(d)}\right)^2 v_{jm}^{(d)}$ term has its own

d	$\mu_{jm}^{(d)}$ Q(15-e_d).e_d	$v_{jm}^{(d)}$ Q(15-i_d).i_d
0	Q6.9	Q0.15
1	Q6.9	Q0.15
2	Q6.9	Q0.15
3	Q6.9	Q0.15
4	Q6.9	Q0.15
5	Q6.9	Q0.15
6	Q6.9	Q0.15
7	Q6.9	Q0.15
8	Q6.9	Q0.15
9	Q5.10	Q0.15
10	Q5.10	Q0.15
11	Q5.10	Q0.15
12	Q7.8	Q0.15
13	Q5.10	Q2.13
14	Q4.11	Q2.13
15	Q4.11	Q2.13
16	Q4.11	Q2.13
17	Q4.11	Q2.13
18	Q4.11	Q2.13
19	Q4.11	Q2.13
20	Q4.11	Q2.13
21	Q4.11	Q2.13
22	Q4.11	Q2.13
23	Q4.11	Q2.13
24	Q3.12	Q3.12
25	Q5.10	Q2.13
26	Q3.12	Q5.10
27	Q3.12	Q5.10
28	Q3.12	Q5.10
29	Q3.12	Q5.10
30	Q3.12	Q5.10
31	Q3.12	Q5.10
32	Q3.12	Q5.10
33	Q3.12	Q4.11
34	Q3.12	Q5.10
35	Q3.12	Q5.10
36	Q2.13	Q5.10
37	Q2.13	Q5.10
38	Q3.12	Q4.11

Table 4.3: Fixed-point formats of $\mu_{jm}^{(d)}$ and $v_{jm}^{(d)}$ for each of the 39 dimensions. These formats are determined by examining all the HMM states trained using the RM1 corpus.

fixed-point format denoted by Q$(47 - 2e_d - i_d).(2e_d + i_d - 16)$, which are 32-bit. It is required to convert each term to the same fixed-point format before summation as shown in Equation (4.10). It is also obvious from the equation that the same fixed-point format should be adopted to C_{jm}, g_{jm} and $\log b_{jm}(\mathbf{o}_t)$ since the arithmetic operation is addition. Furthermore, $\log b_j(\mathbf{o}_t)$ and $\log b_{jm}(\mathbf{o}_t)$ should follow the same format because they are log probabilities. The common fixed-point format of these parameters is denoted by Q(15-p).p, where p is determined from experiments in order to achieve minimum degradation in recognition accuracy. The experimental results are shown in Section 4.4. Table 4.2 summarizes the fixed-point formats discussed in this section.

4.3 Viterbi search

4.3.1 Algorithm

The Viterbi search algorithm has already been discussed in Section 2.3.3. In this section, we focus on the implementation details of the algorithm. In practice, probabilities are converted to their logarithmic forms since multiplication of probabilities becomes addition in the logarithmic domain. The Viterbi likelihood as defined in Equation (2.17) can be expressed as follows in the logarithmic domain.

$$\log V_t(i) = \max_{q_1^{t-1}} \log P(\mathbf{o}_1^t, q_1^{t-1}, q_t = i | W_1^n, \Theta_a) \tag{4.18}$$

Similarly, the initialization and the recursion of the Viterbi algorithm as shown in Equations (2.18) and (2.19) can be converted to their logarithmic forms.

$$\log V_1(i) = \log \pi_i + \log b_i(\mathbf{o}_1) \quad 1 \leq i \leq \Omega \tag{4.19}$$

$$\log V_t(j) = \max_{1 \leq i \leq \Omega} [\log V_{t-1}(i) + \log a_{ij}] + \log b_j(\mathbf{o}_t) \quad 1 \leq j \leq \Omega \tag{4.20}$$

In our implementation of the speech recognizer, a WFST transition is a higher level representation of a sub-word HMM, as discussed in Section 2.6.2. Language model probabilities are incorporated into the search network as WFST transition weights. When a token is leaving from a sub-word HMM and entering into a new sub-word HMM, it is the same as entering into a new WFST transition under the WFST paradigm. The log weight of the new transition is added to the Viterbi likelihood.

$$\log V_t(j) = \max_{1 \leq i \leq \Omega} [\log V_{t-1}(i) + \log z_{ij}] + \log b_j(\mathbf{o}_t) \quad 1 \leq j \leq \Omega \tag{4.21}$$

where z_{ij} is the WFST weight of the new transition. State j is the first state of the HMM on the new WFST transition, whereas State i is the last state of the HMM on the previous WFST transition.

4.3.2 Viterbi search with fixed-point formats

Table 4.4 shows the fixed-point formats of the various data types in the Viterbi algorithm. As shown in (4.19), (4.20) and (4.21), the arithmetic operations on these data types are additions. Therefore, they share the same fixed-point formats as the log emission probability.

Data type	Notation	Fixed-point format
Log emission probability	$\log b_j(\mathbf{o}_t)$	Q(15-p).p
Log initial state probability	$\log \pi_i$	Q(15-p).p
Log HMM transition probability	$\log a_{ij}$	Q(15-p).p
Log WFST transition weight	$\log z_{ij}$	Q(15-p).p
Log Viterbi likelihood	$\log V_t(i)$	Q(15-p).p

Table 4.4: Fixed-point formats of various data types in the Viterbi search

4.4 Recognition accuracy

The framework of floating-point to fixed-point conversion has been presented in the previous sections. In the framework, there are two design variables. They are:

- Fixed-point format of the Hamming-windowed speech samples in feature extraction. The format is 32-bit Q(31-f).f. See Table 4.1.
- Fixed-point format of the log probabilities, which include the log emission probabilities, log HMM transition probabilities, log WFST weights and other related terms, etc. The format is denoted by 16-bit Q(15-p).p. See Table 4.2 and 4.4.

The values of the design variables are determined empirically by experiments. The task under test is the Resource Management (RM1) task [52]. The number of words in the vocabulary is 993. Triphone or biphone HMM models with 3 emitting states and 4 Gaussian mixtures per emitting state are trained on 2880 utterances. Acoustic features are 39-dimensional MFCCs with the zeroth coefficient plus their delta and delta-delta coefficients. The features are extracted from 25ms speech frames with 10ms frame shift. The language model is word-pair grammar (bigram). The recognizer is tested using 1200 utterances. Both floating-point and fixed-point implementations of Juicer are tested.

Table 4.5 shows the word accuracy rate over a range of f and p. By examining the average accuracy in the rightmost column, it shows that the fixed-point system achieves a higher average word accuracy rate when f is between 11 and 13. Similarly, in the bottom row, the average word accuracy rate is higher than the others when p is either 3 or 4. The maximum word accuracy rate achieved is 93.33% when (f, p) is $(13, 3)$ or $(11, 3)$. For this task, the fixed-point system does not show any degradation in recognition accuracy. In fact, the maximum word accuracy rate of the fixed-point system is slightly greater than the floating-point system. We choose $(f, p) = (13, 3)$ as the precision formats of the fixed-point system.

The framework proposed in this chapter can be applied to other speech recognition tasks. The two design variables, namely f and p, can be determined empirically from the test set of a particular speech recognition task. The optimal values of these variables

Fixed-point system						
Q(31-f).f	Q(15-p).p					
	Q10.5	Q11.4	Q12.3	Q13.2	Q14.1	Average
Q16.15	92.78	92.86	92.82	92.74	92.76	92.79
Q17.14	93.23	93.23	93.26	93.21	93.12	93.21
Q18.13	93.28	93.29	93.33	93.22	93.20	93.26
Q19.12	93.26	93.31	93.26	93.15	93.27	93.25
Q20.11	93.28	93.30	93.33	93.22	93.15	93.26
Average	93.17	93.20	93.20	93.11	93.10	

Floating-point (32-bit) system: 93.09%

Table 4.5: Word accuracy rate (%) versus the two design variables, f and p, of the fixed-point implementation of Juicer. The word accuracy rate of the floating-point implementation is also shown. The test set consists of 1200 utterances from the RM1 corpus.

can be found by choosing the combination which demonstrates the greatest word accuracy rate. When these variables are chosen, the fixed-point formats of all the data types can be determined subsequently according to the framework. Specifically, one can follow Table 4.1 to 4.4 to determine the precision format of all the data types in the ASR algorithm.

4.5 Summary

In this chapter, we start from a floating-point implementation of the speech recognizer called Juicer, which normally runs on a desktop platform. Since many embedded platforms do not have a floating-point processing unit, a performance analysis on a fixed-point speech recognizer is necessary. A framework of floating-point to fixed-point conversion is proposed. There are two design variables, which determine the precision of the data types in the framework. The two design variables are found empirically by performing an experiment on the RM1 task. The combination of the design variables with the highest word accuracy rate is chosen. For the RM1 task, the fixed-point implementation does not show any degradation in recognition accuracy. In fact, the word accuracy rate of the fixed-point system is slightly higher than that of the floating-point system. The proposed framework for finding the fixed-point precision format can be applied to other speech recognition tasks.

5
Pure software-based system

To evaluate the real-time performance of different implementations, a pure software-based speech recognizer is developed as the baseline system. The recognizer software is directly ported onto the target platform. Both the floating-point and the fixed-point versions of the recognizer are considered. In this chapter, the architecture of the embedded target platform is first introduced. It is then followed by a timing profile of the speech recognizer.

5.1 Target platform - Altera Nios II processor

The target platform is based on Altera Nios II processor, which is a softcore embedded processor [3]. In contrary to a hardcore processor, a softcore processor allows designers to configure the processor core to suit their application needs. A softcore processor is often described in hardware description language (HDL), which can be synthesized on an FPGA device. In addition, the softcore processor-based approach enables designers to develop the entire system by integrating the processor core with various types of peripherals on a programmable chip (system-on-a-programmable-chip or SOPC). Besides standard peripherals, custom peripherals can also be built. Therefore, a softcore processor-based system offers a flexible platform for hardware-software co-design.

The development board used in this study is Altera Nios II Development Board, Stratix II Edition [2]. The FPGA device is Stratix II EP2S60F672C5ES. The board includes 1MB SRAM and 16MB SDRAM for off-chip memory storage. The system clock frequency is

120MHz.

5.2 System architecture

Figure 5.1 illustrates the architecture of the embedded platform for the pure software baseline system. It is a simple system consisting of a Nios II processor, an SRAM interface and an SDRAM controller, which are connected to the Avalon bus [4]. These intellectual property (IP) blocks are instantiated and downloaded onto the FPGA device.

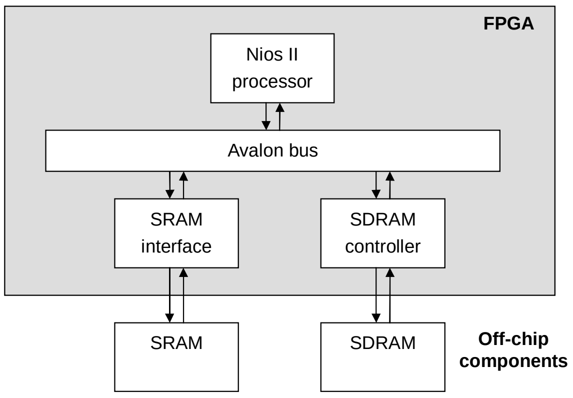

Figure 5.1: Architecture of the embedded platform for implementing the pure software-based speech recognizer

Off-chip memories, for example, SRAM and SDRAM, are connected to their corresponding interfaces. The Nios II processor accesses the external memories by first communicating with the memory interfaces via the Avalon bus. The memory interfaces then generate all the necessary signals for transferring data to and from the external memory chips.

The machine code of the recognizer and all the data structures including the HMM parameters, WFST topology and weights, etc, are stored in the off-chip memories. There are two main reasons for this configuration. First, the size of the internal memory on the FPGA device is simply insufficient for storing all the data structures. Second, this architecture is flexible and extendable to larger vocabulary tasks. If the recognizer requires more memory for larger vocabulary tasks, a simpler and more economical way is to connect larger off-chip memories around the FPGA instead of replacing the original FPGA with a larger device. The underlying architecture and the IP blocks of the system on the FPGA need not to be changed.

5.3 Timing profile

The aim of building a timing profile is to find out where the timing bottleneck is in the speech recognition algorithm. The source code of the speech recognizer is compiled and downloaded to the target platform. The source code is the same as the one tested on the desktop system as described in Chapter 4. It does not use any specialized machine instructions or software routines of the target processor. This ensures that the timing profile obtained here applies to other embedded platforms as well.

A speech utterance from the RM1 corpus is chosen for profiling. The speech recognition algorithm is partitioned into three sub-tasks, namely, *feature extraction*, *GMM emission probability calculation* and *Viterbi search*. The amount of elapsed time in each sub-task is recorded. Table 5.1 shows the timing profile of the pure software-based speech recognizer running on the Nios II platform. Both the floating-point and the fixed-point implementations of the recognizer software are tested on Nios II. The duration of the speech utterance is 2.515s.

	Floating-point		Fixed-point	
	Time(s)	%	Time(s)	%
MFCC feature extraction	1.88	4.45	0.34	7.23
GMM calculation	34.29	81.24	3.24	68.94
Viterbi search	6.04	14.31	1.12	23.83
Total time	42.21	100.00	4.70	100.00
Real-time factor	16.78		1.87	

Table 5.1: Timing profile of running the pure software-based speech recognizer on the Nios II platform. Both the floating-point and the fixed-point versions are considered. The speech utterance duration is 2.515s.

The first point to note from the timing profile is that decoding is much faster in the fixed-point system than the floating-point system. According to the timing profile, the float-point system requires 42.21s for decoding the speech utterance, whereas the fixed-point system only requires 4.70s. The fixed-point system is about 9 times faster than the floating-point system. The timing profile shows that GMM emission probability computation is the major computational bottleneck. For the fixed-point system, about 69% of the decoding time is spent on GMM computation. Viterbi search accounts for about 24% of the total decoding time. Feature extraction is the least computationally intensive. About 7% of the decoding time is spent on it. Thus, design effort should focus on reducing the time spent on GMM computation.

An extended experiment is carried out to examine the timing performance of the fixed-point speech recognizer. In this experiment, the entire test set of the RM1 corpus, which consists of 1200 utterances, is tested on the Nios II platform. The real-time factor of each speech utterance is found. Figure 5.2 shows the experimental result. From the figure, it can be seen that the real-time factor of most of the speech utterances is above 1. The average real-time factor is 1.874, which suggests that the decoding time is about 1.874 times the speech duration on average. This fixed-point pure software-based system does not meet the real-time constraint.

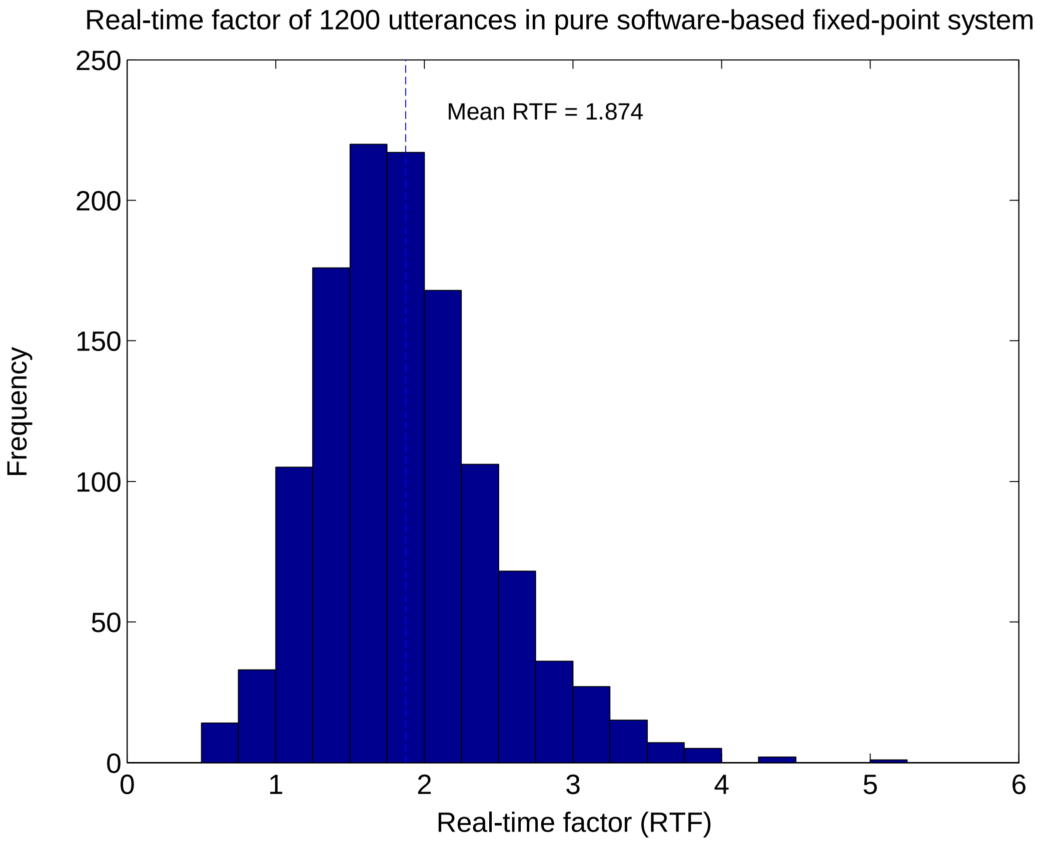

Figure 5.2: Real-time factor of 1200 utterances in the pure software-based fixed-point system. The utterances are from the test set of the RM1 corpus. Pruning beamwidth = 170.

To further analyze the experimental results, another timing measure is introduced. It is called *time delay*, which is the time difference between the speech duration and the decoding time.

$$\text{Time delay} = \text{Decoding time} - \text{Speech duration} \tag{5.1}$$

A *positive* time delay means that the decoding time is longer than the speech duration. Its magnitude is the amount of time needed to wait for the recognizer output after the speech is uttered. For practical applications, it is desirable to build a system, which can generate the word transcription instantly after the speech is uttered. Therefore, a system is a better system when the time delay is getting closer to 0. A *negative* time delay means that the decoding time is shorter than the speech duration. It suggests that the word transcription can be generated instantly after the speech is uttered. Figure 5.3 depicts the time delay of each utterance. Most of the speech utterances have a positive time delay. The average time delay is 2.889s. There is a need for further improvement to achieve real-time performance.

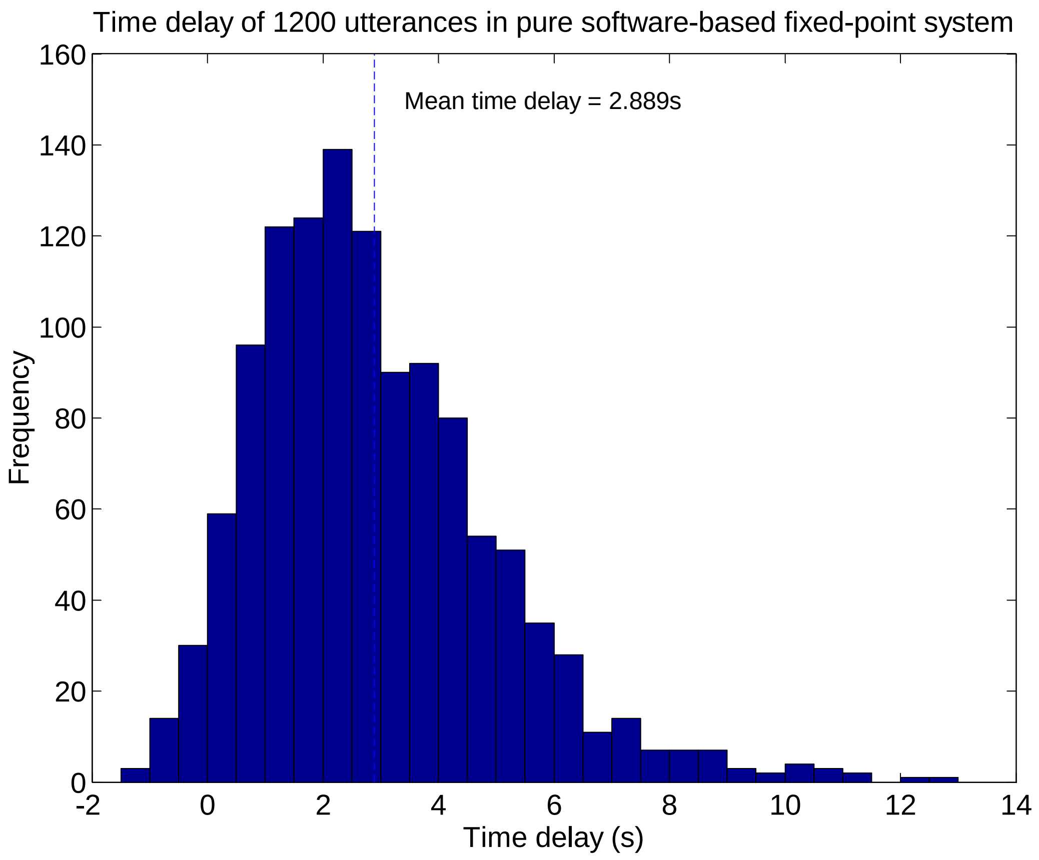

Figure 5.3: Time delay (in seconds) of 1200 utterances in the pure software-based fixed-point system. Pruning beamwidth = 170.

5.4 Summary

In this chapter, a pure software-based speech recognizer is developed on the Nios II platform. The first part of this study is to compare the timing performance between the floating-point implementation and the fixed-point implementation. The experimental results on the RM1 task show that the fixed-point system is about 9 times faster than the floating-point system. The second part of this study is to build a timing profile for the fixed-point system. The results indicate that the pure software-based approach cannot meet the real-time constraints. For the same RM1 task, the real-time factors of most of the speech utterances are greater than 1. Another observation from the timing profile is that the most computationally intensive part of the algorithm is the GMM computation. In order to achieve real-time performance, it is necessary to speed up the GMM computation.

6 Hardware-software co-processing system

To reduce the decoding time, the computation of GMM emission probabilities is separated from the software and implemented in a hardware accelerator.

6.1 System architecture

Figure 6.1 illustrates the architecture of the hardware-software co-processing system. A GMM hardware accelerator is included in the architecture. The hardware accelerator is connected to the Avalon bus. The communication protocol between the accelerator and other modules in the system follows the Avalon bus specification [4]. Hence, the hardware accelerator is a portable IP block, which can be instantiated in other Nios II systems. The accelerator is described in VHDL.

The Nios II processor acts as the control unit of the entire co-processing system. The algorithms of feature extraction and Viterbi search are described in software and run by the Nios II processor. When the system needs to perform an emission probability calculation, the processor instructs the hardware accelerator to carry out the computation. Once the calculation has finished, the processor retrieves the result from the hardware accelerator.

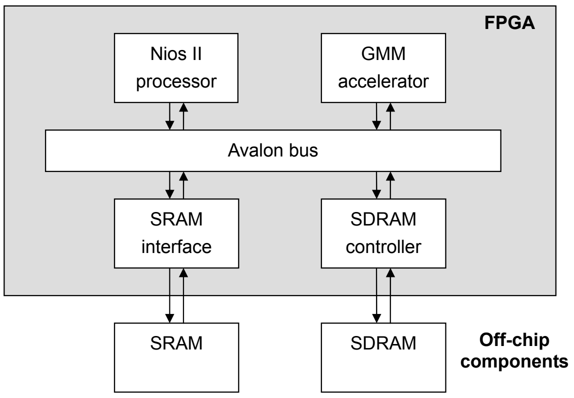

Figure 6.1: System architecture of the hardware-software co-processing recognizer with the GMM hardware accelerator

6.2 GMM emission probability hardware accelerator

6.2.1 Datapath

The GMM hardware accelerator calculates the log emission probability of an observation vector given an HMM state. The algorithm has been discussed in Section 4.2.1. The log emission probability of an observation vector given the m^{th} Gaussian mixture of an HMM state j is shown below.

$$\log b_{jm}(\mathbf{o}_t) = C_{jm} + g_{jm} + \sum_{d=0}^{D-1} \left(o_t^{(d)} - \mu_{jm}^{(d)} \right)^2 v_{jm}^{(d)} \tag{6.1}$$

In the equation, $o_t^{(d)}$ is the d^{th} dimension of the observation vector at time t. D is the dimension of the observation vector, which is typically equal to 39. $\mu_{jm}^{(d)}$ is the d^{th} dimension of the $\boldsymbol{\mu}_{jm}$ mean vector. C_{jm}, $v_{jm}^{(d)}$ and g_{jm} are constants defined as follows.

$$C_{jm} = \log c_{jm} \tag{6.2}$$

$$v_{jm}^{(d)} = \frac{-1}{2\left(\sigma_{jm}^{(d)}\right)^2} \tag{6.3}$$

$$g_{jm} = -\frac{1}{2}\left(D\log(2\pi) + \sum_{d=0}^{D-1} \log \left(\sigma_{jm}^{(d)}\right)^2 \right) \tag{6.4}$$

where c_{jm} is the mixture weight of the m^{th} Gaussian. The symbol $\left(\sigma_{jm}^{(d)}\right)^2$ is the d^{th} feature variance, which is the d^{th} diagonal element of the co-variance matrix.

The log emission probability, $\log b_j(\mathbf{o}_t)$, can be evaluated recursively by the following equation.

$$\log b_j(\mathbf{o}_t) = ((\log b_{j1}(\mathbf{o}_t) \oplus \log b_{j2}(\mathbf{o}_t)) \oplus \log b_{j3}(\mathbf{o}_t)) \oplus ... \oplus \log b_{jM}(\mathbf{o}_t) \tag{6.5}$$

The $\oplus$ symbol represents the *log-add* operator, which has the following definition and approximation.

$$\begin{aligned} x \oplus y &= \log(\exp(x) + \exp(y)) \\ &\approx \begin{cases} y & z < -16 \\ y + \log(1 + \exp(z)) & -16 \leq z < 0 \\ x + \log(1 + \exp(-z)) & 0 \leq z < 16 \\ x & z \geq 16 \end{cases} \end{aligned} \tag{6.6}$$

where $z = x - y$. The number 16 in the above equation serves as a threshold, which is determined empirically by experiments. This approximation does not decrease the recognition accuracy. When $|z|$ is greater than the threshold, the difference between $\exp(x)$ and $\exp(y)$ is large enough to just consider only the greater number. The $\log(1 + \exp(.))$ function can be calculated off-line and stored in a look-up table. The $|z|$ value can be used as the look-up index of the table.

Figure 6.2 illustrates the implementation of the arithmetic unit. At the top, there are 39 parallel computation units. Each one is responsible for one dimension of the observation vector. The results from the 39 units are summed by the parallel adder block. To ensure a large degree of parallelism, the structure of the parallel adder block is similar to an inverted binary tree. After that, g_{jm} and C_{jm} are added to the sum, which gives $\log b_{jm}(\mathbf{o}_t)$ as expressed in (6.1). The accelerator is designed in a pipelined fashion with pipeline registers in between each arithmetic operation. This design allows the GMM parameters ($\mu_{jm}^{(d)}$, $v_{jm}^{(d)}$, g_{jm} and C_{jm}) to be "pumped" into the datapath in consecutive clock cycles. As a result, the $\log b_{jm}(\mathbf{o}_t)$ of each individual Gaussian mixture is outputted consecutively. The *log-add* unit collects all the $\log b_{jm}(\mathbf{o}_t)$ and calculates $\log b_j(\mathbf{o}_t)$ according to (6.5) and (6.6).

Figure 6.3 shows the datapath of the *log-add* unit. The first stage is to find the absolute difference between the two operands. In the second stage, the look-up table stores the value of the $\log(1 + \exp(.))$ function, which is indexed by the absolute difference. It is found that the look-up table requires only 128 16-bit entries. The number of entries is determined as follows. It is shown in Equation (6.6) that the threshold is 16. Since the precision format of the log probability is Q12.3 as discussed in Section 4.4, it only needs to store $16 \times 2^3 = 128$ entries. If the absolute difference is greater than or equal to 16, the most significant 8 bits (Bit 15 to 7) will not be all zeros. In this case, the multiplexer selects

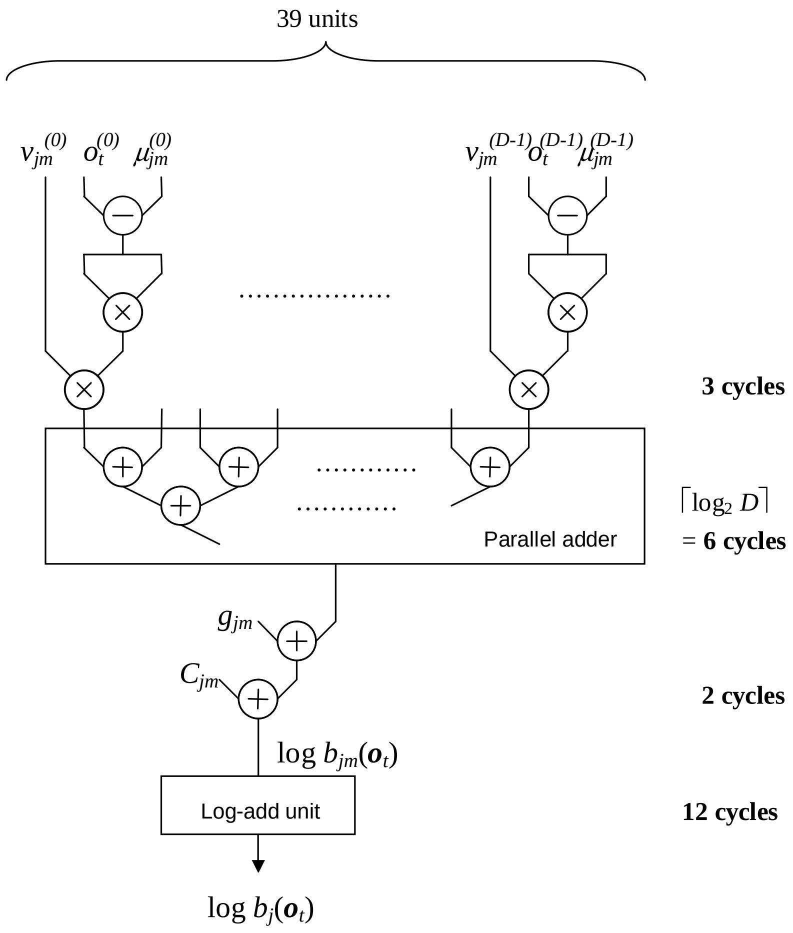
39 units
$v_{jm}^{(0)}$ $o_t^{(0)}$ $\mu_{jm}^{(0)}$
$v_{jm}^{(D-1)}$ $o_t^{(D-1)}$ $\mu_{jm}^{(D-1)}$
3 cycles
Parallel adder
$\lceil \log_2 D \rceil$ = 6 cycles
g_{jm}
C_{jm}
2 cycles
$\log b_{jm}(\boldsymbol{o}_t)$
Log-add unit
12 cycles
$\log b_j(\boldsymbol{o}_t)$

Figure 6.2: Arithmetic unit of the GMM hardware accelerator. The GMM parameters ($\mu_{jm}^{(d)}$, $v_{jm}^{(d)}$, g_{jm} and C_{jm}) are "pumped" into the datapath in consecutive clock cycles. The log emission probability of each Gaussian mixture, $\log b_{jm}(\mathbf{o}_t)$, is passed to the log-add unit consecutively. The final output of the log-add unit is $\log b_j(\mathbf{o}_t)$. Number of clock cycles at each stage is also shown.

the constant zero instead of the look-up value. Otherwise, the value of the $\log(1+\exp(.))$ function is retrieved from the table and passed to the third stage, where the final output is the *log-add* sum of the two operands. This unit receives $\log b_{jm}(\mathbf{o}_t)$ from the previous datapath and accumulates the *log-add* sum of all the Gaussian mixtures.

The number of clock cycles required in each stage of operation is shown in Figure 6.2 and 6.3. Since the inputs of the *log-add* unit are registered, it takes 4 cycles for each *log-add* operation. For 4 Gaussian mixtures, the *log-add* unit performs 3 *log-add* operations and therefore requires 12 cycles. The total number of clock cycles required for calculating $b_j(\mathbf{o}_t)$ is 23.

Memory bandwidth is one of the major issues in an embedded system. As shown in Figure 6.2, the parameters of a Gaussian mixture ($\mu_{jm}^{(d)}$, $v_{jm}^{(d)}$, g_{jm} and C_{jm}) need to be presented to the datapath in each clock cycle in order to avoid any pipeline stalls. Given that the parameters are 16-bit data, it requires to transfer 160 bytes of data to the hardware accelerator per clock cycle. Two main techniques for reducing the memory bandwidth

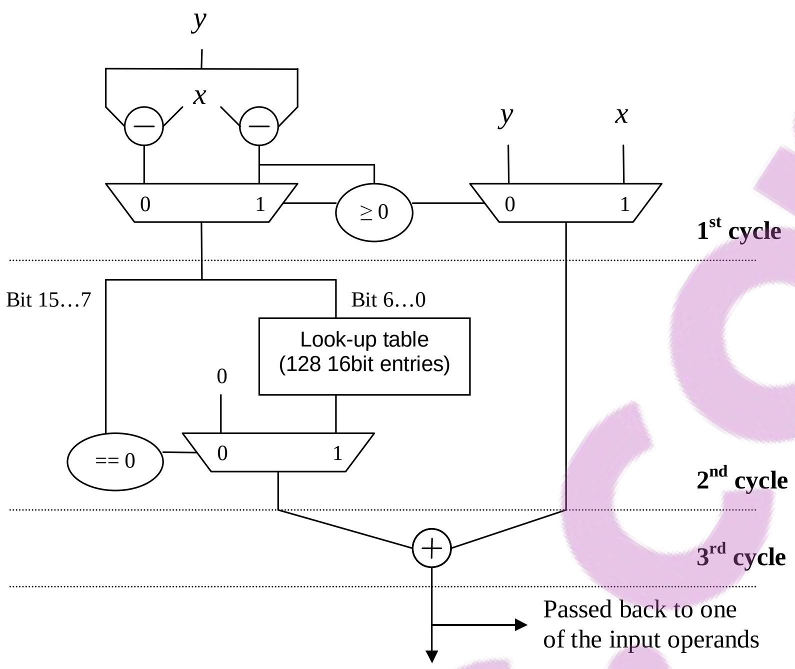

Figure 6.3: Datapath of the log-add unit. It performs the $x \oplus y$ operation, where $\oplus$ is the *log-add* operator. The result can be passed back to one of the input operands for recursive *log-add* operations. This unit accumulates the log probability of each Gaussian mixture, $\log b_{jm}(\mathbf{o}_t)$, and computes the log probability of all the Gaussian mixtures, $\log b_j(\mathbf{o}_t)$.

requirement have been suggested in other literature [31, 62]. The first technique is to adopt a cache inside the hardware accelerator. The purpose of the cache is to store the recently-used model parameters on the FPGA chip instead of reading them from the slow off-chip memory. The disadvantage of this technique is that it requires a large cache in order to observe any useful caching. According to [62], caching only starts to take place when the cache size is increased to 256KB, which is about 40% of the total amount of acoustic parameters. The high percentage of acoustic data that is needed to be cached on the FPGA chip makes the design infeasible for larger vocabulary tasks.

Another technique is to store the acoustic parameters of the N most probable HMM states inside the hardware accelerator [62]. An offline profiling is carried out on the test speech data for finding the N most probable HMM states. This technique suffers similar drawbacks as the first technique. It still requires to store a relatively large amount of acoustic parameters on the FPGA device. Furthermore, the HMM state occupancy statistics is based on the test speech data. In practice, access pattern of the HMM states is highly dependent on the topic of the dialogue. Thus, the statistics collected from the test speech data is not highly representative.

In our proposed system, the acoustic parameters are stored in the off-chip memory modules (SRAM and SDRAM). The hardware accelerator retrieves the parameters from these memory modules when a calculation is requested. To alleviate the memory bandwidth problem, the hardware accelerator adopts a double-buffering scheme. Figure 6.4

illustrates the hardware accelerator with double-buffering. There are two buffers for storing the HMM parameters inside the accelerator. Each buffer contains the acoustic parameters of an HMM state. While the arithmetic unit is reading data from one of the buffers, another buffer is receiving the acoustic parameters of the next HMM state that is needed for calculation from the off-chip memories. By using this scheme, data retrieval and arithmetic operations are running in parallel. In addition, the hardware accelerator only needs to store the parameters of two HMM states, which are about 1280 bytes on the FPGA chip. Observation vector only needs to be loaded once for each speech frame. The size of the observation vector buffer is 78 bytes. Compared with [62], this architecture is independent of the total amount of acoustic parameters in the entire system. It makes the hardware accelerator a portable IP block that can be included in other systems for different vocabulary tasks. Moreover, in our proposed system, there is no assumption on the HMM state occupancy statistics. There is no need to perform any offline profiling.

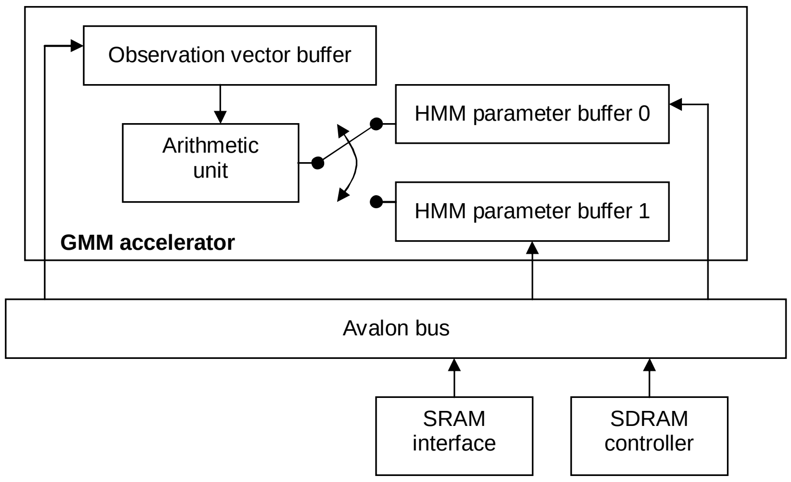

Figure 6.4: Double-buffering inside the GMM hardware accelerator. The arithmetic unit is reading from one buffer while another buffer is retrieving HMM parameters from off-chip memories.

6.2.2 Timing profile

Table 6.1 shows the timing profile of a speech utterance tested on both the pure software-based system and the hardware-software co-processing system. The speech utterance is the same utterance that has been tested previously in Section 5.3. Only the fixed-point implementation is considered since the decoding time is much shorter than the floating-point implementation.

The GMM hardware accelerator greatly shortens the time for GMM computation. It reduces from 3.24s in the pure software-based system to 0.03s in the co-processing system. The speed-up is about 108 times. The total decoding time reduces from 4.70s to 1.49s. The real-time factor improves from 1.87 to 0.59, which is well below 1.00.

	Pure SW-based		HW-SW co-processing	
	Time(s)	%	Time(s)	%
MFCC feature extraction	0.34	7.23	0.34	22.82
GMM calculation	3.24	68.94	0.03	2.01
Viterbi search	1.12	23.83	1.12	75.17
Total time	4.70	100.00	1.49	100.00
Real-time factor	1.87		0.59	

Table 6.1: Timing profile of the pure software-based and the hardware-software co-processing speech recognizer. Data formats are fixed-point. The speech duration is 2.515s.

The hardware-software co-processing system is tested on the entire test set of the RM1 corpus consisting of 1200 utterances. Figure 6.5 compares the real-time factor per utterance between the pure software-based system and the hardware-software co-processing system.

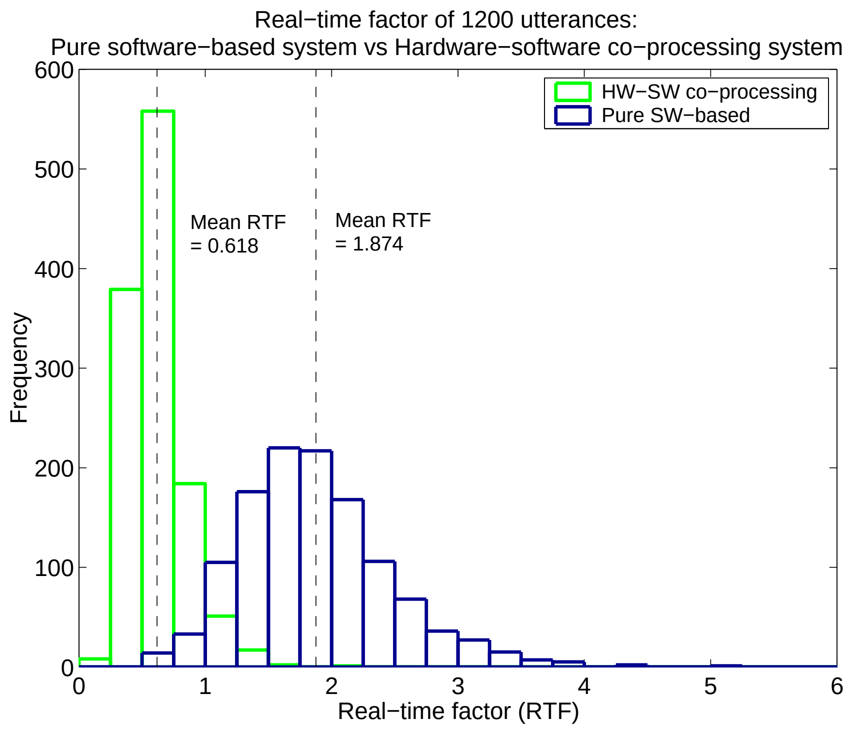

Figure 6.5: Real-time factor of 1200 utterances in two different systems: Pure software-based system versus Hardware-software co-processing system. Pruning beamwidth = 170.

The average real-time factor improves from 1.874 to 0.618. The speed-up is about 3.03 times. The average real-time factor of the hardware-software co-processing system is below 1, which suggests that the decoding time is shorter than the speech duration. The word transcription of a speech utterance can be generated instantly after the utterance has just finished. To study the absolute time difference, the time delay of each utterance is shown in Figure 6.6.

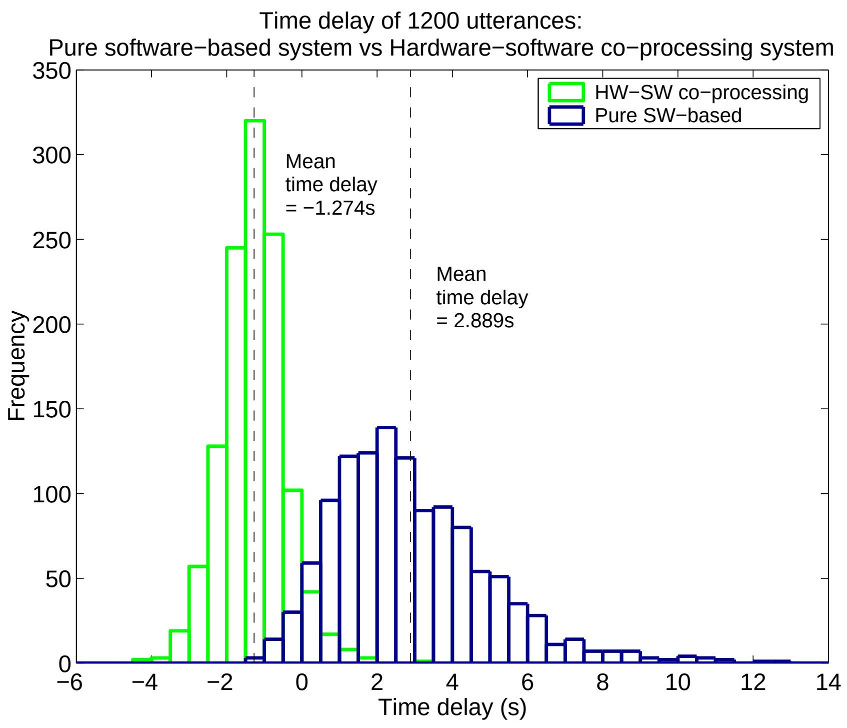

Figure 6.6: Time delay (in seconds) of 1200 utterances in two different systems: Pure software-based system versus Hardware-software co-processing system. Pruning beamwidth = 170.

The average time delay reduces from 2.889s to −1.274s. In the pure software-based system, most of the speech utterances have a positive time delay. In contrast, most of the utterances tested on the hardware-software co-processing system have a negative time delay and therefore meet the real-time constraints. The co-processing system significantly improves the timing performance.

6.2.3 Resource usage

Table 6.2 shows the resource usage of the GMM hardware accelerator. Adaptive Logic Module (ALM) is the building block of a Stratix II FPGA device. Hardware multipliers are also embedded on a Stratix II FPGA. The GMM accelerator consumes 44.1% of the total available ALMs and 59.7% of all the DSP 18-bit × 18-bit multipliers.

	Units	%
Adaptive Logic Module (ALM)	10660	(44.1%)
DSP 18-bit×18-bit multiplier	86	(59.7%)

Table 6.2: Resource usage of the GMM hardware accelerator. The device is Stratix II EP2S60F672C5ES FPGA.

6.3 Adaptive pruning

In Section 6.2, a hardware-software co-processing architecture with a GMM emission probability hardware accelerator is presented. The co-processing system demonstrates a significant improvement on the decoding speed. Figure 6.5 and Figure 6.6 show that out of the 1200 utterances in the RM1 corpus, about 94.08% of them have a real-time factor of less than 1. In this section, we aim at further increasing this percentage.

In general, the real-time factor of each utterance is different. It is because the number of *active HMM states* (or *tokens*) is different for different utterances. *Beam pruning* controls the number of active states by setting a probability threshold [44, 43]. If there are no clear winning HMM states, the active states will have similar likelihood scores. There will be a large number of active states above the pruning threshold. The recognizer will then need more time to iterate through the entire set of active states.

One method for lowering the number of active states is to adopt a tighter pruning beamwidth. In this case, the number of active states above the pruning threshold will be smaller. However, it will introduce search errors, which often decrease the recognition accuracy. Our goal is to reduce the decoding time of those utterances that have a real-time factor of greater than 1, while keeping the recognition accuracy of other utterances. In order to fulfil our goal, an *adaptive* pruning scheme is proposed. In this scheme, the pruning beamwidth is adaptive according to the number of active HMM states.

6.3.1 Algorithm

Figure 6.7 shows the pseudocode of the speech recognition algorithm with adaptive pruning. In the beginning, the beamwidth is initialized to a value (Line 4). Before token-passing, the algorithm modifies the pruning beamwidth according to the number of active tokens, $n(\tilde{Q}_t)$. If the number of tokens is greater than a threshold, N_{upper}, it means that the recognizer needs to go through a relatively large number of active states during token-passing. A tighter beamwidth is necessary. The beamwidth is decreased by a certain amount denoted by δ (Line 11 - 12). However, if the number of active tokens is smaller than another threshold, N_{lower}, the workload for the recognizer is small. Therefore, if the beamwidth is tightened previously, it will be relaxed and its value will be increased by δ (Line 13 - 16). The rest of the algorithm is the same as the one shown in Figure 2.5.

The proposed pruning scheme is more flexible than the narrow and fixed pruning scheme. During decoding of an utterance, the number of active tokens is time-varying. The fixed pruning scheme applies a tight beamwidth throughout the entire utterance regardless of the number of active tokens. On the other hand, the adaptive pruning scheme allows relaxation of the beamwidth when the workload becomes less heavy. It introduces fewer search errors than the narrow and fixed beam pruning scheme.

In terms of implementation, the proposed scheme is simpler than another pruning technique called histogram pruning [58]. In histogram pruning, the recognizer only allows at most N number of active tokens to stay in the search space. If there are more tokens than the allowable limit, the N most probable tokens will remain active. Other tokens will be pruned. Implementing histogram pruning requires a sorted list of the token scores. For each token, the recognizer needs to perform an insertion sort. It needs to search for the token's ranking in a sorted list of the previously-recorded token scores. Maintaining the tokens in a sorted order is computationally intensive. In contrast, the adaptive pruning scheme is simpler. It only requires to record the number of active tokens and a few decision-making statements (if-statements) for adjusting the beamwidth at the end of each speech frame (Line 11 - 17).

Algorithm 3 Speech recognition algorithm with adaptive beam pruning

```
/* $\tilde{Q}_t$ is a set of HMM states having a hypothesis at time $t$ */
$\tilde{Q}_1 \leftarrow Q_{word-start}$
$score_{q,1} \leftarrow 0$ for all $q \in \tilde{Q}_1$
$pruning\_beamwidth \leftarrow original\_pruning\_beamwidth$

for $t = 1$ to $T$ do
    $\mathbf{o}_t \leftarrow Feature\_extraction(Frame_t)$

    $max\_score \leftarrow \max(score_{q,t})$ for all $q \in \tilde{Q}_t$

    if $n(\tilde{Q}_t) > N_{upper}$ then
        $pruning\_beamwidth \leftarrow pruning\_beamwidth - \delta$
    else if $n(\tilde{Q}_t) < N_{lower}$ then
        if $pruning\_beamwidth < original\_pruning\_beamwidth$ then
            $pruning\_beamwidth \leftarrow pruning\_beamwidth + \delta$
        end if
    end if

    $pruning\_threshold \leftarrow max\_score - pruning\_beamwidth$
    $\tilde{Q}_{t+1} \leftarrow \{\}$

    for all $q \in \tilde{Q}_t$ do
        if $score_{q,t} > pruning\_threshold$ then
            $log\_emis\_prob \leftarrow Emission\_prob\_calc(\mathbf{o}_t, q)$
            $\mathcal{V} \leftarrow Viterbi\_search(log\_emis\_prob, q, t)$
            $\tilde{Q}_{t+1} \leftarrow \tilde{Q}_{t+1} \cup \mathcal{V}$
        end if
    end for
end for

$\mathcal{Q} \leftarrow \tilde{Q}_{T+1} \cap Q_{word-end}$
$best\_hypothesis \leftarrow \operatorname{argmax}_{q \in \mathcal{Q}}(score_{q,T+1})$
```

Figure 6.7: Speech recognition algorithm with adaptive beam pruning

6.3.2 Timing profile

Figures 6.8 and 6.9 illustrate the real-time factor and the time delay of the co-processing system respectively. Fixed beam pruning and adaptive beam pruning are compared. The beamwidth is held constant at 170 for the fixed beam pruning scheme. In adaptive beam pruning, the *original_beamwidth* variable is also set to 170. The thresholds, N_{lower} and N_{upper}, are 1900 and 2300 respectively. The beamwidth adjustment value is 10 ($\delta = 10$). These parameters are determined empirically.

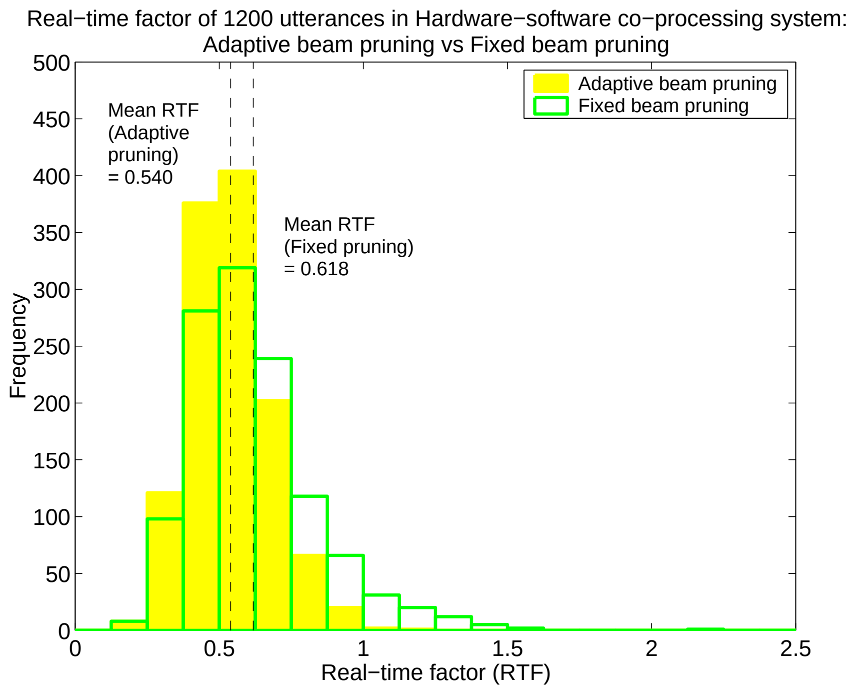

Figure 6.8: Real-time factor of 1200 utterances in Hardware-software co-processing system: Adaptive beam pruning versus Fixed beam pruning.

Adaptive beam pruning reduces the number of utterances having a real-time factor above 1. In the fixed beam pruning scheme, about 94.08% of the utterances have a real-time factor below 1. When the adaptive beam pruning scheme is used, this percentage increases to 99.75%. Only 3 out of 1200 utterances have a real-time factor above 1. Compared with the fixed beam pruning scheme, there is a small degradation in recognition accuracy. The word accuracy rate decreases from 93.33%, as observed in Section 4.4, to 93.16% when the adaptive pruning scheme is adopted. We have also tried to tighten the adaptive pruning scheme by adjusting N_{upper} and N_{lower} to smaller values ($N_{upper} = 1700$, $N_{lower} = 1250$), so that the real-time factors of all the utterances are below 1. The word accuracy rate reduces to 92.62%.

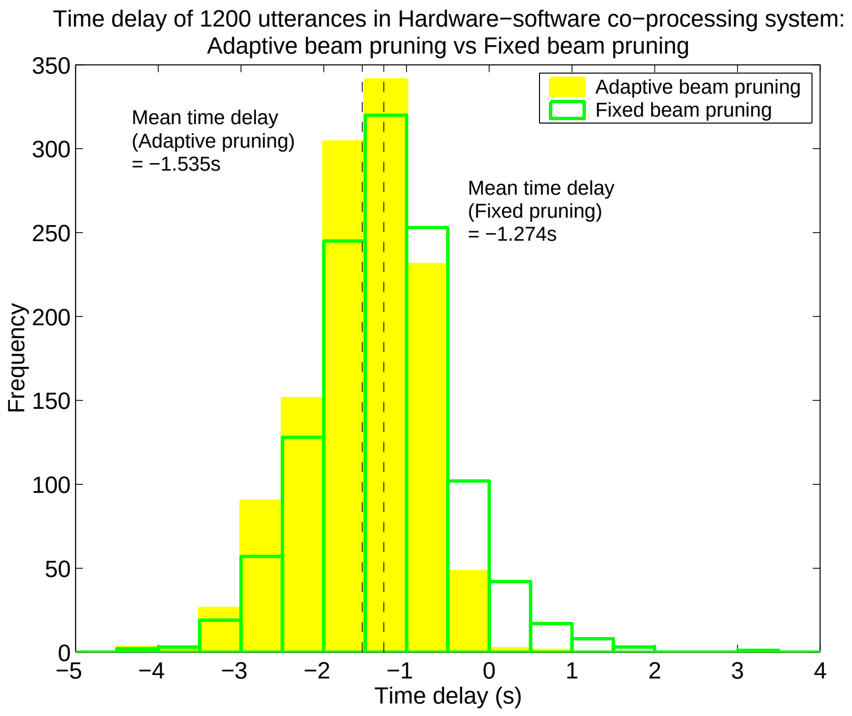

Figure 6.9: Time delay (in seconds) of 1200 utterances in Hardware-software co-processing system: Adaptive beam pruning versus Fixed beam pruning.

6.4 Performance evaluation

Table 6.3 compares the performance of our proposed system with other existing systems. Our proposed system belongs to the *hardware-software co-processing system* category. Two versions of our system are shown. One version is the system with fixed beam pruning. Another version is the one with adaptive beam pruning.

The first observation is that the clock frequency of the pure software-based systems is higher than the systems in the other two categories. This is mainly because algorithmic parallelism is possible in both pure hardware-based systems and hardware-software co-processing systems. Hence, the number of clock cycles required for performing a task is greater in pure software-based systems. As a result, a higher clock frequency is needed. On the other hand, pure hardware-based systems and hardware-software co-processing systems can run the same task in lower clock frequency, which is a clear advantage over the pure software-based systems.

The first performance metric is word accuracy rate. As shown in the table, the word accuracy rate of our proposed system is within the range of other published systems. Our proposed system performs better than PocketSphinx [26] and In Silico Vox [32, 33] systems. For the AT&T system [9], the recognition accuracy is slightly better than our proposed system. However, in their publication, it shows that the acoustic features are stored in files. The StrongARM platform accesses the acoustic feature files from a PC

System	CPU/Platform	Clock frequency (MHz)	Word accuracy rate (%)	Real-time factor
Pure software-based system				
PocketSphinx [26]	StrongARM	206	86.05	0.87
AT&T [9]	StrongARM	206	≈ 94	1.00
Pure hardware-based system				
In Silico Vox [32, 33]	Dedicated design on Xilinx Virtex-II Pro XC2VP30 FPGA	50	89.10	2.30
Speech Silicon [57]	Dedicated design on Xilinx Virtex-4 XCE4VSX35 FPGA	100	N/A	N/A
Hardware-software co-processing system				
Seoul National University [31, 62]	MicroBlaze on Xilinx Virtex-4 XC4VSX35 FPGA	100	96.20[1]	N/A
Our proposed system with fixed beam pruning	Nios II on Altera Stratix II EP2S60F672C5ES FPGA	120	93.33	0.62
Our proposed system with adaptive beam pruning	Nios II on Altera Stratix II EP2S60F672C5ES FPGA	120	93.16	0.54

[1]The test set contains only 300 utterances.

Table 6.3: Performance of recently developed embedded speech recognition systems and our proposed systems on the 993-word RM1 task

via an Ethernet link. It suggests that the acoustic features may not be generated by the StrongARM platform. Their timing profile may not include the time for feature extraction. Therefore, in order to achieve a real-time factor of 1.00 *including* also feature extraction, their system may need a tighter beamwidth than the one suggested in the publication. The actual word accuracy rate will be lower if a tighter beamwidth is applied. For the Seoul National University [31, 62] system, the word accuracy rate is higher than other existing systems. In spite of that, the test set of their experiments contains only 300 utterances, whereas there are 1200 test utterances in our experiments. Thus, there is no direct comparison in terms of word accuracy rate.

The second performance metric is real-time factor. From the table, it shows that the real-time factor of our proposed system is better than other reported systems. The average real-time factor of our system is well below 1.00. This is necessary since each utterance has a different real-time factor. As illustrated in Figure 6.8, there is a range of real-time factors within the entire corpus. It is better to have an average real-time factor well below 1.00 so that the real-time factors of most of the speech utterances are below 1.00.

Our proposed system is a hardware-software co-processing system. It is worthwhile to compare our system with the Seoul National University system [31, 62], which is also under the same category. Here is a list of major differences. Some of them have been discussed in Section 6.2.

- To deal with the memory bandwidth issue, the GMM accelerator in [31, 62] uses an internal cache to store the recently accessed HMM parameters. Their study shows that caching only starts to take place when the cache size is increased to 256KB, which is about 40% of the total amount of acoustic parameters. This makes the design infeasible for larger vocabulary tasks. Our system adopts a *double-buffering* scheme, in which the GMM accelerator retrieves the parameters of the next HMM state from off-chip memory in the background while performing calculation of the current HMM state. The amount of parameters that is required to be stored inside the accelerator is kept to minimum. This scheme avoids any pipeline stalls and makes the accelerator a portable IP block for larger vocabulary tasks.

- Their second solution to the memory bandwidth issue is to store the parameters of the N most probable HMM states inside the GMM accelerator. To find the N most probable HMM states, another experiment is carried out to record the access pattern from the test set of the corpus. In practical applications, the access pattern is related to the topic of the current conversation and therefore varies in different situations. Thus, the access pattern observed from the test set can be very different from the practical situations. In our approach, we do not make any assumptions on the access pattern of the HMM parameters.

- In their design of the GMM accelerator, there is only one pipeline for calculating one dimension of a Gaussian mixture. In our proposed system, we further explore parallelism in GMM calculation and include 39 parallel pipelines for calculating the 39 dimensions of a Gaussian mixture simultaneously.

- In their system, there are two hardware accelerators - a GMM accelerator and a Viterbi accelerator. In our system, there is only one GMM accelerator, which deals with the most computationally intensive part of the algorithm. The timing profile suggests that this architecture is sufficient for our target application.

- The real-time factor of their system is not shown, whereas we have shown the real-time factor of our proposed system in detail.

6.5 Summary

In Chapter 5, we have shown that the pure software-based fixed-point recognizer does not meet the real-time constraint on the Nios II platform. It shows that GMM emission probability calculation is the major computational bottleneck. In this chapter, a hardware-software co-processing system, in which a hardware accelerator is designed to speed up the GMM computation, is proposed.

The hardware accelerator uses double-buffering to retrieve the acoustic parameters from off-chip memory and calculates the emission probability in parallel. In the RM1 experiments, the word accuracy rate is 93.33% and the average real-time factor is 0.62, which is about 3 times faster than the pure software-based system. About 94.08% of all the test utterances in the RM1 corpus have a real-time factor below 1.00. To further increase this percentage, an adaptive beam pruning algorithm is developed. The algorithm applies tighter pruning when the number of active hypotheses exceeds a certain threshold. For the same RM1 task, the word accuracy rate is 93.16%. The real-time factor further reduces to 0.54. The percentage of the utterances which have a real-time factor smaller than 1.00 increases to 99.75%.

The performance of the proposed co-processing system is compared with other reported systems. The recognition accuracy of our system is within the range of the other systems. In terms of the decoding speed, the real-time factor of our system is smaller (better) than those of the other systems. In this chapter, we have also mentioned the major differences between our proposed system and [31, 62] which also adopts a hardware-software co-processing approach.

7 Dynamic composition of WFST

7.1 Motivation

In Section 2.6.2, the weighted finite state transducer (WFST) approach for search space representation is described. In the application of WFST in ASR, the idea is to represent each individual knowledge source by a WFST and fully integrate them into a unified WFST by the composition algorithm [36, 35, 50]. The search space is fully expanded. The composition of knowledge sources is a one-off process and is done offline. Therefore it is often referred to *static* composition.

There are two main advantages with the static approach. First, the decoder design is simple because all the knowledge sources are integrated into one compact WFST. The knowledge sources are *decoupled* from the Viterbi search and therefore the decoder does not need to perform any combination of knowledge sources during decoding. The second advantage is that the fully integrated transducer can be further optimized by algorithms, such as, determinization, minimization and weight-pushing [36, 35].

Despite the above advantages, there are several drawbacks with the static approach. They include:

- It does not allow on-line modification of knowledge sources once they have been fully integrated.
- The composition and optimization of the fully integrated WFST has prohibitively

high memory requirement when the constituent WFSTs are large and complex.

- The size of the fully integrated WFST can be very large, resulting in large memory requirement during decoding.

One way of addressing these issues is to perform *dynamic* transducer composition during decoding. Instead of representing the entire search space by an optimized transducer, it is possible to factorize the search space into two or more transducers. These component transducers are built statically and optimized separately. The combination is done dynamically during decoding. In this dynamic approach, on-line modification of knowledge sources is allowed. For example, by factoring out the language model transducer from the search space, the ASR system can update and adapt the language model probabilities during its operation.

In this chapter, we investigate several existing dynamic composition approaches and propose our improved algorithm, which avoids the creation of non-coaccessible transitions, performs weight look-ahead and does not impose any constraints to the topology of component WFSTs. The chapter is organized as follows. Section 7.2 briefly describes static WFST composition and how a fully integrated WFST is generated. Section 7.3 gives a general overview on current approaches to dynamic WFST composition. Section 7.4 describes our dynamic composition algorithm. Experimental results on different composition methods are shown in Section 7.5. Finally, Section 7.6 summarizes the findings.

7.2 Static WFST composition in ASR

In Chapter 3, the theory of static WFST composition has already been discussed. Static WFST composition involves integrating all the knowledge sources into one WFST. It can be represented by the following expression [38].

$$T = \pi_\epsilon(min(det(\tilde{H} \circ det(\tilde{C} \circ det(\tilde{L} \circ G))))) \tag{7.1}$$

In the above expression, $\tilde{H}$ represents the HMM topology; $\tilde{C}$ is a WFST which maps context-dependent phones to context-independent phones; $\tilde{L}$ is the lexicon WFST and G is the language model (LM) WFST. The symbol $\circ$ is the composition operator. Transducer optimization algorithms, for example determinization and minimization, are represented by *det* and *min* operators respectively. The $\tilde{}$ symbol means that the WFST is augmented with auxiliary symbols which are necessary for the success of transducer optimization. The π_ϵ operation replaces the auxiliary symbols by ϵ (null) symbols. The final transducer T is a fully integrated transducer which maps HMM state sequences to word sequences.

Figure 7.1 illustrates an example of WFST composition, $\tilde{L} \circ G$, over the tropical semiring. The states of $\tilde{L} \circ G$ are labelled with a (S_1, S_2) pair, where S_1 and S_2 are the state indices of $\tilde{L}$ and G respectively. The composition algorithm starts from the initial states of both WFSTs. Any ϵ-output transitions of $\tilde{L}$ and any ϵ-input transitions of G are treated as "free-entries", which do not need to match with any other transitions. For example, the two transitions from state 0 of $\tilde{L}$ are ϵ-output transitions. In G, the $\epsilon : \epsilon/0.1$ transition is an ϵ-input transition. Without the need to match with other transitions, these transitions are copied to the $\tilde{L} \circ G$ transducer.

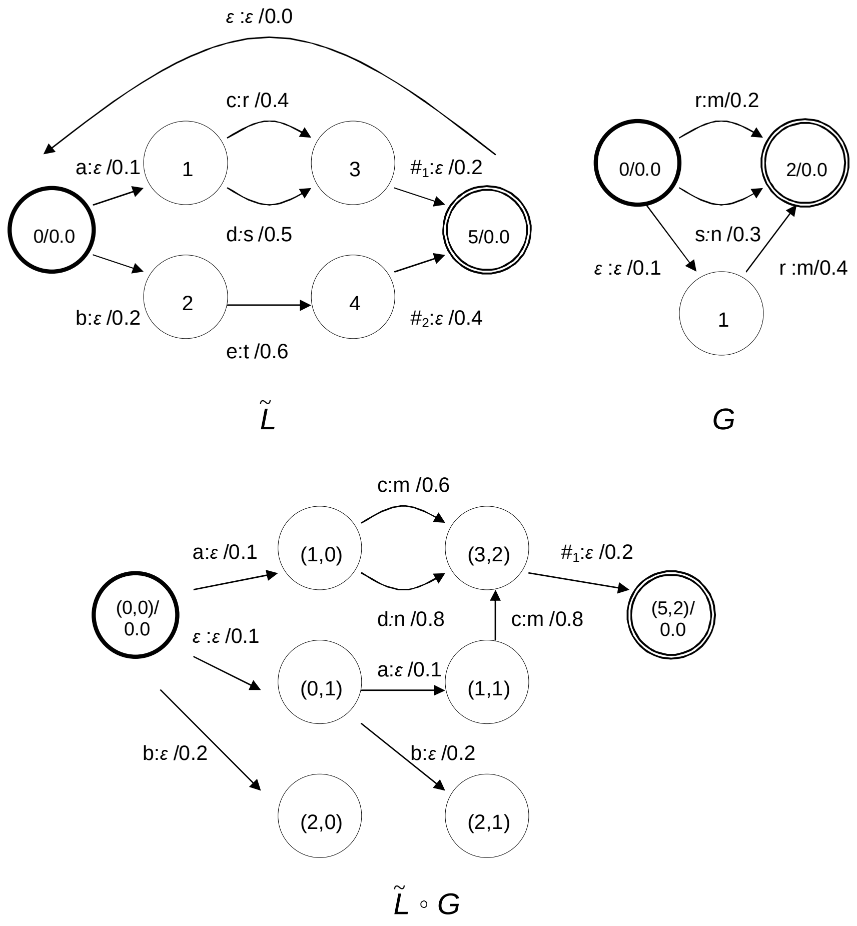

Figure 7.1: Static composition of $\tilde{L} \circ G$ over the tropical semiring. In $\tilde{L}$, #1 and #2 are the auxiliary word-end markers.

The idea behind composition is to match the output symbols of $\tilde{L}$ with the input symbols of G. Figure 7.2 shows the pseudocode of the composition algorithm. The aim of the algorithm is to determine a set of transitions emanating from state (S_1, S_2) of the composed WFST. The first step is to duplicate all the "free-entries" transitions to the composed WFST. These transitions include ϵ-input transitions from S_2 (Line 2-9) and ϵ-output transitions from S_1 (Line 11-16). The *dest*() function finds the destination state of a transition. The *new_state* variable stores the destination state of the newly composed transition. Duplication of ϵ-input transitions from S_2 is allowed only when S_1 is an *anchor* state. This avoids redundant duplication of these transitions in the composed WFST [50]. In the example, the initial state of $\tilde{L}$ is the anchor state.

Algorithm 4 $composed\_trans\_set \leftarrow WFST\_composition(S_1, S_2)$

```
1: $composed\_trans\_set \leftarrow \{\}$
2: if $S_1$ is an anchor state (initial state) then
3:    for all $\epsilon$-input transitions, $trans2$, emanating from state $S_2$ do
4:       $new\_trans \leftarrow trans2$
5:       $new\_state \leftarrow (S_1, dest(trans2))$
6:       $new\_trans\_with\_dest \leftarrow (new\_trans, new\_state)$
7:       $composed\_trans\_set \leftarrow composed\_trans\_set \cup \{new\_trans\_with\_dest\}$
8:    end for
9: end if
10:
11: for all $\epsilon$-output transitions, $trans1$, emanating from state $S_1$ do
12:    $new\_trans \leftarrow trans1$
13:    $new\_state \leftarrow (dest(trans1), S_2)$
14:    $new\_trans\_with\_dest \leftarrow (new\_trans, new\_state)$
15:    $composed\_trans\_set \leftarrow composed\_trans\_set \cup \{new\_trans\_with\_dest\}$
16: end for
17:
18: for all non-$\epsilon$-output transitions, $trans1$, emanating from state $S_1$ do
19:    for all non-$\epsilon$-input transitions, $trans2$, emanating from state $S_2$ do
20:       if $out\_sym(trans1) == in\_sym(trans2)$ then
21:          $new\_weight \leftarrow weight(trans1) \otimes weight(trans2)$
22:          $new\_trans \leftarrow in\_sym(trans1) : out\_sym(trans2)/new\_weight$
23:          $new\_state \leftarrow (dest(trans1), dest(trans2))$
24:          $new\_trans\_with\_dest \leftarrow (new\_trans, new\_state)$
25:          $composed\_trans\_set \leftarrow composed\_trans\_set \cup \{new\_trans\_with\_dest\}$
26:       end if
27:    end for
28: end for
29: return $composed\_trans\_set$
```

Figure 7.2: Pseudocode of the *WFST_composition* function. S_1 and S_2 are the state indices of the two constituent WFSTs which are deterministic.

The next step is to examine all the non-ϵ-output transitions from S_1 and all the non-ϵ-input transitions from S_2 (Line 18 and 19). For each pair of $trans1$ and $trans2$, the algorithm determines whether the output symbol of $trans1$ matches with the input symbol of $trans2$ (Line 20). If there is a match, the weight of the composed transition is equal to the semiring-product ($\otimes$) of the weights of $trans1$ and $trans2$ (Line 21). The input symbol and the output symbol of the composed transition are $in\_sym(trans1)$ and $out\_sym(trans2)$ respectively (Line 22). The new_state is a pair of state indices, where the first and the second indices are the destination states of $trans1$ and $trans2$ respectively (Line 23).

7.3 Current Approaches to Dynamic WFST Composition

Several groups of researchers have proposed different approaches to dynamic WFST composition. They include Dolfing [20], Willett [61], Caseiro [14, 12, 11, 13] and Hori [25]. The first step of any dynamic composition algorithm is to factorize the entire search space into two or more constituent WFSTs before decoding. Approaches include:

1. Separating the entire G from other knowledge sources, resulting in two WFSTs [14];
2. Separating only part of the G (G_i or so called the *incremental* LM) from other knowledge sources. The remaining part of the LM (G_s or the *smearing* LM) is statically composed with other knowledge sources, resulting in two WFSTs [20, 61];
3. Factorizing the entire search space into multiple WFSTs [25].

During decoding, the constituent WFSTs are composed on-the-fly. There are two main approaches for combining constituent transducers dynamically, namely *with no lookahead* and *with lookahead.*

The *no lookahead* approach is basically the dynamic version of static WFST composition. The term “dynamic” means that the search space is expanded on-the-fly during decoding. The active search space is time-varying and depends on the number of distinct word histories amongst all the active tokens. The search space is expanded only if there are active tokens. In the previous example as illustrated in Figure 7.1, active tokens reside in the $\tilde{L}$ transducer. In order to keep a record of the word history, each of the active tokens stores an attribute, which is basically the state index of the G transducer (that is S_2). The search space is dynamically expanded according to the location of the tokens and their S_2 state indices. For example, suppose there is a token leaving the $a : \epsilon/0.1$

transition and passing through state 1 of $\tilde{L}$. If the S_2 attribute of the token is 0, the dynamic composition algorithm will try to match the symbols between state 1 of $\tilde{L}$ and state 0 of G. In other words, it is equivalent to calling the *WFST_composition*$(1, 0)$ function.

There are two problems with this approach. The first problem is the creation of non-coaccessible transitions or so called "dead-end" transitions [14]. They are the transitions which will not reach the final state of a transducer. In Figure 7.1, the two $b : \epsilon/0.2$ transitions in $\tilde{L} \circ G$ are non-coaccessible transitions. These transitions are created because they are ϵ-output transitions in $\tilde{L}$. Therefore, they are copied to the composite transducer. However, the output symbol of the next $\tilde{L}$ transition $e : t/0.6$ is t, which cannot be matched with any input symbols in G. As a result, these transitions cannot go further and become "dead-end" transitions. The generation of these "dead-end" transitions increases the number of redundant transitions and tokens in the search space.

The second problem is the delay of the application of transducer weights. Weights in G are not applied to the composite transducer until there is an actual mapping between the output symbols and the input symbols of the component transducers (Line 21 - 22 in Figure 7.2). For pruning efficiency, it is beneficial to introduce the G weights as early as possible before the actual mapping of symbols occurs.

The *lookahead* approach proposed by Caseiro [14] addresses the above problems. He subdivides $\tilde{L}$ into two regions, a *prefix* region and a *suffix* region. The prefix region is the region between the initial state of $\tilde{L}$ and the non-ϵ-output transitions. In Figure 7.3, the prefix region is bounded by the grey rectangle. The region between the non-ϵ-output transitions and the final state is the suffix region, which is bounded by the white rectangle. A suffix region is ended with word-end transitions ($\#_1$ and $\#_2$).

A set of *anticipated* output symbols for each ϵ-output transitions is built inside the prefix region. Anticipated symbols are all the possible output symbols that can be encountered eventually along the current path in the transducer. For example, the $a : \epsilon/0.1$ transition in $\tilde{L}$ has two anticipated output symbols, r and s, since the path can go to either $c : r/0.4$ or $d : s/0.5$, where r and s are their output symbols respectively. The function of the anticipated symbol sets is to provide some lookahead information. An ϵ-output transition in $\tilde{L}$ will be expanded in the composition only if there is a match between its anticipated symbol set and the input labels of G.

Figure 7.4 shows the pseudocode of the lookahead approach. Basically, there are four cases that need to be considered. The first case is about the ϵ-input transitions of G (Line 2-10). The treatment is the same as the no lookahead approach, in which these ϵ-input transitions are duplicated to the composite transducer when S_1 is an anchor state. The second case is about the ϵ-output transitions in the prefix region of $\tilde{L}$ (Line 13-16). These transitions are treated by the $WFST\_dynamic\_prefix()$ function which will be discussed later. The third case is about the ϵ-output transitions in the suffix region of $\tilde{L}$ (Line

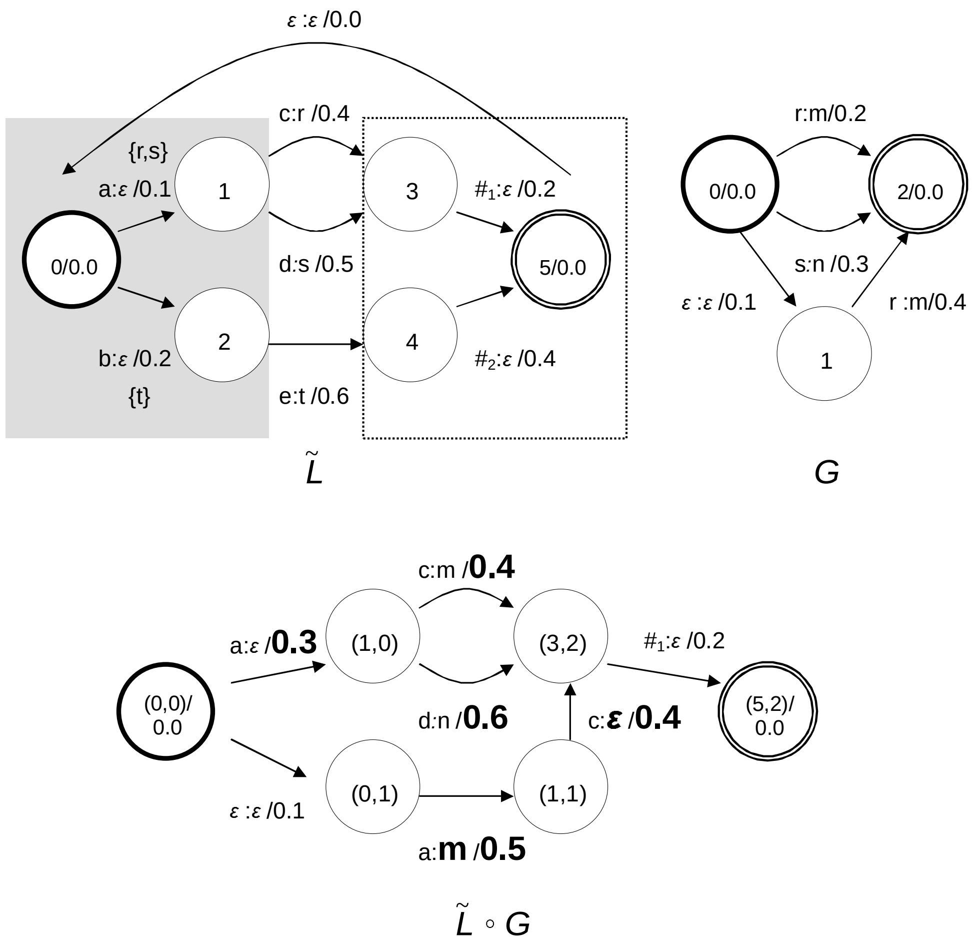

Figure 7.3: The lexicon WFST ($\tilde{L}$) in Caseiro's approach. The $\tilde{L}$ transducer is partitioned into two regions - a prefix region (grey rectangle) and a suffix region (white rectangle). $\#_1$ and $\#_2$ are word-end markers. $\{\}$ indicates an anticipated output symbol set. Two constituent WFSTs, $\tilde{L}$ and G, are composed using Caseiro's approach. The bold symbols and weights in $\tilde{L} \circ G$ indicate the changes compared with the *no lookahead* approach.

18-22). In this approach, these transitions are treated as "free-entries" and duplicated in the composite transducer (Line 19). The last case is about the non-ϵ-output transitions in $\tilde{L}$. The $WFST\_dynamic\_non\_eps()$ function, which will also be described later, deals with these transitions.

The pseudocode of the $WFST\_dynamic\_prefix()$ function is shown in Figure 7.5. The first step is to decide whether there are any matches between symbols (Line 3-8). For each transition emanating from S_2 of G (that is, $trans2$), the algorithm examines whether the input symbol of $trans2$ can be found in the anticipated output symbol set of $trans1$ (Line 4). The weights of all the matched $trans2$ transitions are semiring-added ($\oplus$) to give the lookahead weight (Line 5). Over the tropical semiring, the $\oplus$ operation is the min operator. Hence, it is equivalent to finding the minimum weight from all the matched $trans2$ transitions.

Algorithm 5 $composed\_trans\_set \leftarrow WFST\_dynamic\_compose\_caseiro(S_1, S_2)$

```
1: $composed\_trans\_set \leftarrow \{\}$
2: **if** $S_1$ is an anchor state (initial state) **then**
3:   **for all** $\epsilon$-input transitions, $trans2$, emanating from state $S_2$ **do**
4:     /* $\epsilon$-input $trans2$ */
5:     $new\_trans \leftarrow trans2$
6:     $new\_state \leftarrow (S_1, dest(trans2))$
7:     $new\_trans\_with\_dest \leftarrow (new\_trans, new\_state)$
8:     $composed\_trans\_set \leftarrow composed\_trans\_set \cup \{new\_trans\_with\_dest\}$
9:   **end for**
10: **end if**
11:
12: **for all** $\epsilon$-output transitions, $trans1$, emanating from state $S_1$ **do**
13:   **if** $trans1$ is in the prefix region **then**
14:     /* $\epsilon$-output $trans1$ in the prefix region */
15:     $trans\_set \leftarrow WFST\_dynamic\_prefix(trans1, S_1, S_2)$
16:     $composed\_trans\_set \leftarrow composed\_trans\_set \cup trans\_set$
17:   **else**
18:     /* $\epsilon$-output $trans1$ in the suffix region */
19:     $new\_trans \leftarrow trans1$
20:     $new\_state \leftarrow (dest(trans1), S_2)$
21:     $new\_trans\_with\_dest \leftarrow (new\_trans, new\_state)$
22:     $composed\_trans\_set \leftarrow composed\_trans\_set \cup \{new\_trans\_with\_dest\}$
23:   **end if**
24: **end for**
25:
26: /* Non-$\epsilon$-output $trans1$ */
27: **for all** non-$\epsilon$-output transitions, $trans1$, emanating from state $S_1$ **do**
28:   $trans\_set \leftarrow WFST\_dynamic\_non\_eps(trans1, S_1, S_2)$
29:   $composed\_trans\_set \leftarrow composed\_trans\_set \cup trans\_set$
30: **end for**
31: **return** $composed\_trans\_set$
```

Figure 7.4: Pseudocode of the *WFST_dynamic_compose_caseiro* function. S_1 and S_2 are the state indices of the two constituent WFSTs which are deterministic.

After symbol-matching, three cases are considered. In the first case, there are no matched symbols (Line 9-11). It means that the $trans1$ transitions cannot match with any $trans2$ transitions from S_2. It is the "dead-end" situation, and therefore there is no need to expand.

The second case is that there are more than 1 matched symbols (Line 12-20). This is the case where the actual symbol cannot be determined; however, the lookahead weight can be incorporated into the composite transducer. First, $\Delta lookahead\_weight$ is found by *semiring-division*, which is opposite to the $\otimes$ operation (Line 14). Semiring-division is basically substraction over the tropical semiring. Hence, it is the same as calculating the difference between the new lookahead weight and the previous lookahead weight in (S_1, S_2). This idea is similar to finding the *residual* probability in language model lookahead as discussed in Section 2.6.1. The $\Delta lookahead\_weight$ is semiring-multiplied by the weight of $trans1$ (Line 15). Accordingly, a newly-composed transition is determined (Line 16). Note that the output symbol of this composed transition is ϵ since the actual symbol is indeterministic. The lookahead information is stored in a table for future reference (Line 19-20).

The last case is that there is only 1 matched symbol (Line 22-39). In this scenario, the output symbol can be determined. If it is the first time to obtain only 1 matched symbol, the output symbol of the composed transition is emitted (Line 27). The weight of the transition is found similarly as in the previous case (Line 25-26). However, if it is not the first time, the output symbol has already been emitted. The lookahead procedure has finished; and therefore, $trans1$ is duplicated to the composed transducer (Line 34).

Figure 7.6 shows the pseudocode of the $WFST\_dynamic\_non\_eps()$ function, which deals with the non-ϵ-output transitions in $\tilde{L}$. There are three differences between this function and the $WFST\_dynamic\_prefix()$ function. First, symbol-matching is done on the non-ϵ output symbol of $trans1$ (Line 4) since this transition has a real output symbol and there is no need to consider a set of anticipated symbols. Second, the outcome of symbol-matching is either no match or only 1 match. There are no multiple matches. Third, the S_2 index of new_state is changed to $dest(matched\_trans)$ (Line 19 and 26).

To summarize, the lookahead approach addresses two major issues of the non-lookahead approach. The differences between these two approaches are shown graphically in Figure 7.1 and 7.3. First, it avoids the creation of non-coaccessible transitions. Second, the lookahead approach allows early application of G weights before encountering the actual non-ϵ output symbols in $\tilde{L}$. This idea is similar to language model lookahead and WFST weight-pushing.

Algorithm 6 $trans\_set \leftarrow WFST\_dynamic\_prefix(trans1, S_1, S_2)$

```
1: $new\_lookahead\_weight \leftarrow \bar{0}$
2: $matched\_trans \leftarrow \{\}$
3: for all non-$\epsilon$-input transitions, $trans2$, emanating from state $S_2$ do
4:    if $in\_sym(trans2) \in anticipated\_out\_sym\_set(trans1)$ then
5:       $new\_lookahead\_weight \leftarrow new\_lookahead\_weight \oplus weight(trans2)$
6:       $matched\_trans \leftarrow matched\_trans \cup \{trans2\}$
7:    end if
8: end for
9: if $matched\_trans == \{\}$ then
10:    /* "Dead-end" transitions. Ignore. */
11:    $trans\_set \leftarrow \{\}$
12: else if $n(matched\_trans) > 1$ then
13:    /* More than 1 match */
14:    $\Delta lookahead\_weight \leftarrow lookahead\_weight(S_1, S_2)^{-1} \otimes new\_lookahead\_weight$
15:    $new\_weight \leftarrow \Delta lookahead\_weight \otimes weight(trans1)$
16:    $new\_trans \leftarrow in\_sym(trans1) : \epsilon/new\_weight$
17:    $new\_state \leftarrow (dest(trans1), S_2)$
18:    $trans\_set \leftarrow \{(new\_trans, new\_state)\}$
19:    $finish\_lookahead(new\_state) \leftarrow false$
20:    $lookahead\_weight(new\_state) \leftarrow new\_lookahead\_weight$
21: else
22:    /* Only 1 match */
23:    if $finish\_lookahead(S_1, S_2) == false$ then
24:       /* First time to get only 1 match */
25:       $\Delta lookahead\_weight \leftarrow lookahead\_weight(S_1, S_2)^{-1} \otimes new\_lookahead\_weight$
26:       $new\_weight \leftarrow \Delta lookahead\_weight \otimes weight(trans1)$
27:       $new\_trans \leftarrow in\_sym(trans1) : out\_sym(matched\_trans)/new\_weight$
28:       $new\_state \leftarrow (dest(trans1), S_2)$
29:       $trans\_set \leftarrow \{(new\_trans, new\_state)\}$
30:       $finish\_lookahead(new\_state) \leftarrow true$
31:       $lookahead\_weight(new\_state) \leftarrow new\_lookahead\_weight$
32:    else
33:       /* Already got only 1 match before */
34:       $new\_trans \leftarrow trans1$
35:       $new\_state \leftarrow (dest(trans1), S_2)$
36:       $trans\_set \leftarrow \{(new\_trans, new\_state)\}$
37:       $finish\_lookahead(new\_state) \leftarrow true$
38:       $lookahead\_weight(new\_state) \leftarrow new\_lookahead\_weight$
39:    end if
40: end if
41: return $trans\_set$
```

Figure 7.5: Pseudocode of the *WFST_dynamic_prefix* function. $trans1$ is one of the ϵ-output transitions from S_1 in the prefix region. S_2 is the state index of the second constituent transducer.

Algorithm 7 $trans\_set \leftarrow WFST\_dynamic\_non\_eps(trans1, S_1, S_2)$

```
 1: $new\_lookahead\_weight \leftarrow \bar{0}$
 2: $matched\_trans \leftarrow \{\}$
 3: for all non-$\epsilon$-input transitions, $trans2$, emanating from state $S_2$ do
 4:   if $in\_sym(trans2) == out\_sym(trans1)$ then
 5:     $new\_lookahead\_weight \leftarrow new\_lookahead\_weight \oplus weight(trans2)$
 6:     $matched\_trans \leftarrow matched\_trans \cup \{trans2\}$
 7:   end if
 8: end for
 9: if $matched\_trans == \{\}$ then
10:   /* "Dead-end" transitions. Ignore. */
11:   $trans\_set \leftarrow \{\}$
12: else
13:   /* Only 1 match */
14:   if $finish\_lookahead(S_1, S_2) == false$ then
15:     /* First time to get only 1 match */
16:     $\Delta lookahead\_weight \leftarrow lookahead\_weight(S_1, S_2)^{-1} \otimes new\_lookahead\_weight$
17:     $new\_weight \leftarrow \Delta lookahead\_weight \otimes weight(trans1)$
18:     $new\_trans \leftarrow in\_sym(trans1) : out\_sym(matched\_trans)/new\_weight$
19:     $new\_state \leftarrow (dest(trans1), dest(matched\_trans))$
20:     $trans\_set \leftarrow \{(new\_trans, new\_state)\}$
21:     $finish\_lookahead(new\_state) \leftarrow true$
22:     $lookahead\_weight(new\_state) \leftarrow new\_lookahead\_weight$
23:   else
24:     /* Already got only 1 match before */
25:     $new\_trans \leftarrow trans1$
26:     $new\_state \leftarrow (dest(trans1), dest(matched\_trans)))$
27:     $trans\_set \leftarrow \{(new\_trans, new\_state)\}$
28:     $finish\_lookahead(new\_state) \leftarrow true$
29:     $lookahead\_weight(new\_state) \leftarrow new\_lookahead\_weight$
30:   end if
31: end if
32: return $trans\_set$
```

Figure 7.6: Pseudocode of the *WFST_dynamic_non_eps* function. $trans1$ is one of the non-ϵ-output transitions from S_1. S_2 is the state index of the second constituent transducer.

7.4 Proposed Approach to Dynamic WFST Composition

We base our approach on that of Caseiro. Specifically, two component WFSTs are built: $(\tilde{C}_{opt} \circ \tilde{L}_{opt})$ and G, where $\tilde{C}_{opt}$ and $\tilde{L}_{opt}$ are $min(det(\tilde{C}))$ and $min(det(\tilde{L}))$ respectively. Component transducers are combined with *look-ahead*, avoiding the creation of "dead-end" transitions. Early application of G weights is also performed.

There are however two major differences between our approach and Caseiro's approach. In [14], he presented a specialized algorithm to compose $\tilde{L}$ and G. He made two assumptions (or constraints) about his approach. They are:

- $\tilde{L}$ is an acyclic graph, apart from the loop which connects the final state of $\tilde{L}$ to the initial state (Figure 7.3)
- No weight look-ahead is performed in the suffix region.

While the first assumption holds for a typical lexicon, it is not true for an arbitrary WFST. For example, the $(\tilde{C}_{opt} \circ \tilde{L}_{opt})$ WFST is cyclic in general. For the second assumption, no weight look-ahead is performed in the suffix region. However, in order to achieve better pruning efficiency, weights should be distributed or "pushed" to the initial state as far as possible. Hence, look-ahead of weights, as well as the avoidance of non-coaccessible transitions, should also be performed in the suffix region.

In the following subsections, we describe how the anticipated output label sets are found in the $(\tilde{C}_{opt} \circ \tilde{L}_{opt})$ transducer. We also describe how this transducer is dynamically composed with G during decoding [16].

7.4.1 Finding the Anticipated Output Labels

The entire $(\tilde{C}_{opt} \circ \tilde{L}_{opt})$ transducer is subdivided into *prefix* regions. Each prefix region is terminated with non-ϵ output symbol transitions. All the other transitions are ϵ-output transitions.

Figure 7.7 illustrates an example of a cyclic $(\tilde{C}_{opt} \circ \tilde{L}_{opt})$ transducer. For simplicity, only the output symbols are shown. The transducer is segmented into three prefix regions. Each of them is ended with non-ϵ output symbol transitions. The anticipated output label set can be found by a simple depth-first traversal algorithm.

Symbol and weight lookahead is performed in each prefix region. Since each prefix region is followed by another prefix region, there is no suffix region. The lookahead algorithm always tries to distribute the weights to the initial state as far as possible.

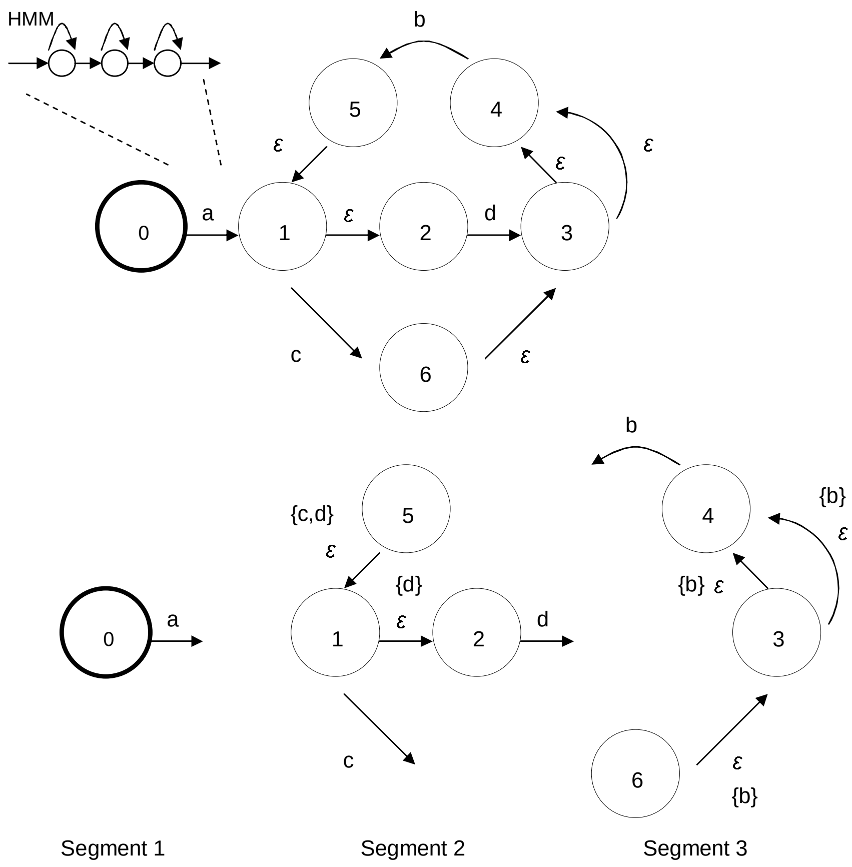

Figure 7.7: A $(\tilde{C}_{opt} \circ \tilde{L}_{opt})$ WFST is segmented into *prefix* regions, where symbol and weight lookahead is performed. For simplicity, only the output symbols are shown. Each ϵ-output transition has an anticipated output symbol set which is denoted by $\{...\}$. Each transition of the $(\tilde{C}_{opt} \circ \tilde{L}_{opt})$ transducer is substituted by an HMM.

7.4.2 The Dynamic Composition Algorithm

The dynamic composition algorithm follows a token-passing paradigm [64]. In dynamic composition, tokens reside in the $(\tilde{C}_{opt} \circ \tilde{L}_{opt})$ transducer. Each WFST transition is substituted by an HMM as illustrated in Figure 7.7. A *token* has the following attributes.

- *token.accWeight* stores the accumulated weight of a token. In the token-passing algorithm, when two tokens meet at the same location in the search space, the one with a lower *accWeight* remains [1].

- *token.*S_1 is the state index of the $(\tilde{C}_{opt} \circ \tilde{L}_{opt})$ transducer where the token is located. The token is ready to be passed into the transitions emanating from S_1. Depending on the lookahead result, some of the transitions could be "dead-end". The token would not be passed into these "dead-end" transitions.

[1]Weights in WFST are negative log probabilities over both the log and tropical semirings (Weight = -log prob). Hence, a lower weight means a greater probability.

- *token.S_2* is the state index of the G transducer to which the token is referenced. It is the word history of the token.
- *token.path_record* stores the output symbols emitted during token-passing. Since the output symbols of G are words, the *path_record* attribute stores a transcription of the recognized words.
- *token.finish_lookahead* is a boolean attribute which indicates whether lookahead has finished in the current prefix region.
- *token.lookahead_weight* stores the already-applied lookahead weight.

The pseudocode of our proposed algorithm is shown in Figure 7.8. The algorithm examines all the transitions from S_1 and determines whether the transitions are "dead-end". Since there is no suffix region, only two cases need to be considered - ϵ-output transitions (Line 4-7) and non-ϵ-output transitions (Line 10-13). The algorithm calls two helper functions in Line 5 and 11.

Algorithm 8 $token\_set \leftarrow WFST\_dynamic\_compose\_proposed(token)$

```
1: $token\_set \leftarrow \{\}$
2: $S_1 \leftarrow token.S_1$
3: /* $\epsilon$-output $trans1$ in the prefix region */
4: for all $\epsilon$-output transitions, $trans1$, emanating from state $S_1$ do
5:     $new\_token\_set \leftarrow WFST\_dynamic\_proposed\_prefix(trans1, token)$
6:     $token\_set \leftarrow token\_set \cup new\_token\_set$
7: end for
8:
9: /* Non-$\epsilon$-output $trans1$ */
10: for all non-$\epsilon$-output transitions, $trans1$, emanating from state $S_1$ do
11:     $new\_token\_set \leftarrow WFST\_dynamic\_proposed\_non\_eps(trans1, token)$
12:     $token\_set \leftarrow token\_set \cup new\_token\_set$
13: end for
14: return $token\_set$
```

Figure 7.8: Pseudocode of our proposed algorithm. It tries to propagate a $token$ at S_1 of the $(\tilde{C}_{opt} \circ \tilde{L}_{opt})$ transducer.

The $WFST\_dynamic\_proposed\_prefix()$ function deals with ϵ-output transitions. It checks whether the token can be passed to the $trans1$ transition. The function is similar to Caseiro's approach. The difference is that in our proposed approach, lookahead information is stored in the token. The first step is symbol-matching (Line 5-10). There are four cases: no match (Line 11-13), more than one match (Line 14-21), first time to get one match (Line 24-32) and already got one match before (Line 34-39). The token is passed to $trans1$ only if there is at least one match. The token's $accWeight$ is updated by $\Delta lookahead\_weight$ and $weight(trans1)$ (Line 17, 27 and 35)[2]. If this is the first time to obtain only one match, the output symbol of the matched $trans2$ can be enqueued to the token's path record (Line 30).

The second helper function is the $WFST\_dynamic\_proposed\_non\_eps()$ function, which deals with non-ϵ-output transitions. The first step is symbol-matching (Line 5-10). There are two possibilities: no match (Line 11-13) or 1 match (Line 15-39). If there is one match, the token's $accWeight$ and $path\_record$ are updated depending on whether it is the first time to obtain only 1 match (Line 16-24). In addition, the token also reaches the end of a prefix region. This is because each prefix region is ended with non-ϵ-output transitions. In order to prepare the token for the next prefix region, the lookahead attributes of the token is reset (Line 25-26). The S_2 attribute is also updated (Line 27).

The ϵ-input transitions of G are treated in this helper function (Line 31-39). If an ϵ-input transition from S_2 is found, the token is duplicated with a new S_2 attribute (Line 36). The algorithm continues to check the new S_2 state and duplicates the token until no more ϵ-input transition is found. The $accWeight$ of the token is updated in each iteration (Line 32). The $path\_record$ is also extended if the output symbol of the transition is non-ϵ (Line 33-35).

7.5 Experimental Results

The aim of this experiment is to compare the performance and the resource requirements of our dynamic composition algorithm with other dynamic composition approaches and the static approach. The following list briefly describes the different approaches under test.

Static Perform decoding on the integrated ($opt(\tilde{C}_{opt} \circ \tilde{L}_{opt} \circ G)$).

Dynamic (Incremental, no look-ahead) Introduce unigram probabilities to build ($\tilde{C}_{opt} \circ \tilde{L}_{opt} \circ G_{uni}$). Dynamically compose this WFST with $G_{tri-uni}$, which is a trigram deviation from unigram, during decoding *without look-ahead* (i.e. no control on non-coaccessible paths and no weight look-ahead).

Dynamic (Caseiro) Build ($\tilde{C}_{opt} \circ \tilde{L}_{opt}$) and G WFSTs. Dynamically compose them during decoding. Since the topology of ($\tilde{C}_{opt} \circ \tilde{L}_{opt}$) is different from $\tilde{L}$ as in his approach, there is no direct comparison. To simulate his method, the control of non-coaccessible paths and weight look-ahead is prohibited until the token reaches a word-end marker inside a prefix region. This *no look-ahead* region can be considered

[2]In Line 35, it is unnecessary to semiring-multiply the $accWeight$ by $\Delta lookahead\_weight$ since $\Delta lookahead\_weight$ must equal to $\bar{1}$.

Algorithm 9 $new\_token\_set \leftarrow WFST\_dynamic\_proposed\_prefix(trans1, token)$

```
1: $new\_lookahead\_weight \leftarrow \bar{0}$
2: $matched\_trans \leftarrow \{\}$
3: $S_1 \leftarrow token.S_1$
4: $S_2 \leftarrow token.S_2$
5: for all non-$\epsilon$-input transitions, $trans2$, emanating from state $S_2$ do
6:    if $in\_sym(trans2) \in anticipated\_out\_sym\_set(trans1)$ then
7:       $new\_lookahead\_weight \leftarrow new\_lookahead\_weight \oplus weight(trans2)$
8:       $matched\_trans \leftarrow matched\_trans \cup \{trans2\}$
9:    end if
10: end for
11: if $matched\_trans == \{\}$ then
12:    /* “Dead-end” transitions. Ignore. */
13:    $new\_token\_set \leftarrow \{\}$
14: else if $n(matched\_trans) > 1$ then
15:    /* More than 1 match */
16:    $\Delta lookahead\_weight \leftarrow token.lookahead\_weight^{-1} \otimes new\_lookahead\_weight$
17:    $token.accWeight \leftarrow token.accWeight \otimes \Delta lookahead\_weight \otimes weight(trans1)$
18:    $token.finish\_lookahead \leftarrow false$
19:    $token.lookahead\_weight \leftarrow new\_lookahead\_weight$
20:    Pass the token to $trans1$
21:    $new\_token\_set \leftarrow \{token\}$
22: else
23:    /* Only 1 match */
24:    if $token.finish\_lookahead == false$ then
25:       /* First time to get only 1 match */
26:       $\Delta lookahead\_weight \leftarrow token.lookahead\_weight^{-1} \otimes new\_lookahead\_weight$
27:       $token.accWeight \leftarrow token.accWeight \otimes \Delta lookahead\_weight \otimes$
          $weight(trans1)$
28:       $token.finish\_lookahead \leftarrow true$
29:       $token.lookahead\_weight \leftarrow new\_lookahead\_weight$
30:       Enqueue $out\_sym(matched\_trans)$ to $token.path\_record$
31:       Pass the token to $trans1$
32:       $new\_token\_set \leftarrow \{token\}$
33:    else
34:       /* Already got only 1 match before */
35:       $token.accWeight \leftarrow token.accWeight \otimes weight(trans1)$
36:       $token.finish\_lookahead \leftarrow true$
37:       $token.lookahead\_weight \leftarrow new\_lookahead\_weight$
38:       Pass the token to $trans1$
39:       $new\_token\_set \leftarrow \{token\}$
40:    end if
41: end if
42: return $new\_token\_set$
```

Figure 7.9: Pseudocode of the *WFST_dynamic_proposed_prefix*() function. It checks whether the $token$ can be passed to the $trans1$ transition.

Algorithm 10 $new\_token\_set \leftarrow WFST\_dynamic\_proposed\_non\_eps(trans1, token)$

```
1: $new\_lookahead\_weight \leftarrow \bar{0}$
2: $matched\_trans \leftarrow \{\}$
3: $S_1 \leftarrow token.S_1$
4: $S_2 \leftarrow token.S_2$
5: for all non-$\epsilon$-input transitions, $trans2$, emanating from state $S_2$ do
6:    if $in\_sym(trans2) == out\_sym(trans1)$ then
7:       $new\_lookahead\_weight \leftarrow new\_lookahead\_weight \oplus weight(trans2)$
8:       $matched\_trans \leftarrow matched\_trans \cup \{trans2\}$
9:    end if
10: end for
11: if $matched\_trans == \{\}$ then
12:    /* "Dead-end" transitions. Ignore. */
13:    $new\_token\_set \leftarrow \{\}$
14: else
15:    /* Only 1 match */
16:    if $token.finish\_lookahead == false$ then
17:       /* First time to get only 1 match */
18:       $\Delta lookahead\_weight \leftarrow token.lookahead\_weight^{-1} \otimes new\_lookahead\_weight$
19:       $token.accWeight \leftarrow token.accWeight \otimes \Delta lookahead\_weight \otimes weight(trans1)$
20:       Enqueue $out\_sym(matched\_trans)$ to $token.path\_record$
21:    else
22:       /* Already got only 1 match before */
23:       $token.accWeight \leftarrow token.accWeight \otimes weight(trans1)$
24:    end if
25:    $token.finish\_lookahead \leftarrow false$
26:    $token.lookahead\_weight \leftarrow \bar{1}$
27:    $token.S_2 \leftarrow dest(matched\_trans)$
28:    Pass the token to $trans1$
29:    $new\_token\_set \leftarrow \{token\}$
30:    /* Duplicate the token for $\epsilon$-input transitions from the new $S_2$ state */
31:    while there is an $\epsilon$-input transition, $trans2$, emanating from $token.S_2$ do
32:       $token.accWeight \leftarrow token.accWeight \otimes weight(trans2)$
33:       if $out\_sym(trans2)$ is non-$\epsilon$ then
34:          Enqueue $out\_sym(trans2)$ to $token.path\_record$
35:       end if
36:       $token.S_2 \leftarrow dest(trans2)$
37:       Pass the token to $trans1$
38:       $new\_token\_set \leftarrow new\_token\_set \cup \{token\}$
39:    end while
40: end if
41: return $new\_token\_set$
```

Figure 7.10: Pseudocode of the *WFST_dynamic_proposed_non_eps*() function.

as the *suffix* region as in his method. Look-ahead resumes after the token has passed the word-end marker.

Dynamic (Our approach) Build $(\tilde{C}_{opt} \circ \tilde{L}_{opt})$ and G WFSTs. Dynamically compose them as described in Section 7.4.

The performance of different approaches was assessed using the Wall Street Journal (WSJ1) corpus [48]. The number of words in the vocabulary was 20000. Cross-word triphone HMM models were trained on the "si_tr_s" set of 38275 utterances using 39-dimensional PLPs. A trigram language model, with 19979 unigrams, 3484372 bigrams and 2949590 trigrams, was used to test the development test set "si_dt_20" from WSJ1 database, consisting of 503 utterances. The experiment was carried out on a PC platform (AMD processor running at 2GHz). The speech recognizer was Juicer.

Figure 7.11 shows the word error rate (WER) against the real-time factor (RTF) of different approaches. Each point on the curve corresponds to the WER versus RTF characteristics for a particular beamwidth. The WERs are similar to those obtained from the previous studies [42]. In comparison with other dynamic composition approaches, the curve of the proposed approach is closer to the origin of the plot, which suggests that the proposed method shows better WER versus RTF performance. One important observation is that the proposed method significantly outperforms the other two dynamic approaches at narrow and moderately-wide beamwidths. For example, at the level of 17% WER (moderately-wide beam), the RTF of our approach is about 65% and 48% of the *no look-ahead* approach and Caseiro's method respectively. This confirms that look-ahead is necessary for good accuracy-time tradeoff in narrow and moderately-wide beamwidth scenarios.

Comparing our approach with static composition, the WERs are similar at each pruning setting. It suggests that our approach is close to the WFST optimization performed during static composition. At the same level of 17% WER, the RTF of the proposed approach is about 60% more than the RTF of static composition. This is due to the overhead, for example, matching symbols in lookahead, searching tokens in a list, etc, required during dynamic composition.

Figure 7.12 illustrates the RTF against the average number of tokens per frame. Our approach has a steeper slope in the figure, which indicates that it requires more time to process each token than the static case. Also it can be seen that the other two dynamic approaches have a lot more tokens per frame than both our approach and the static approach, which shows that the avoidance of non-coaccessible transitions in our approach helps to reduce the number of redundant tokens.

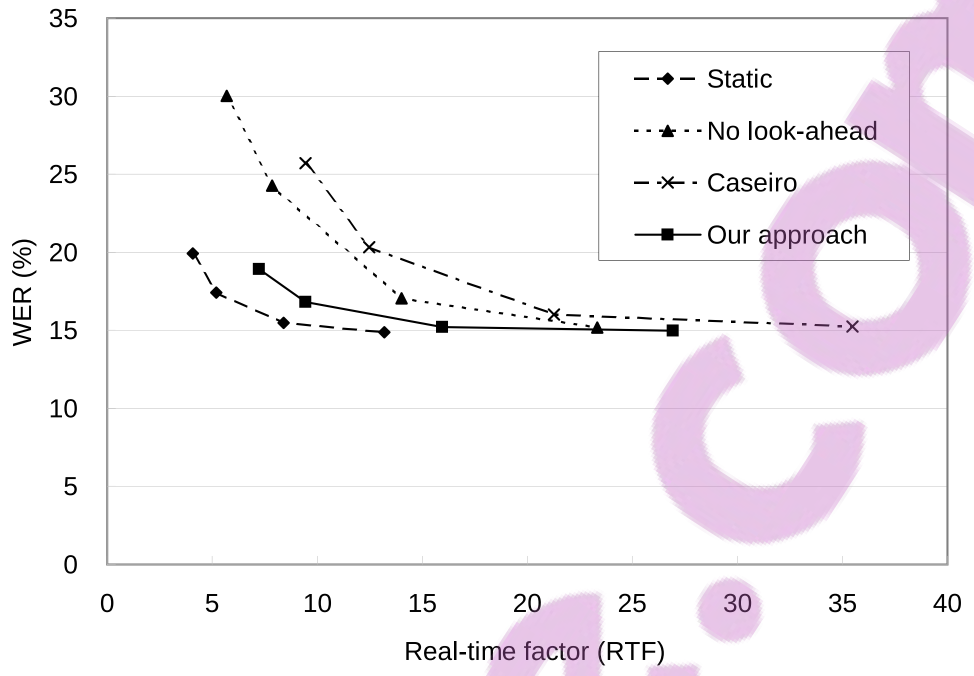

Figure 7.11: WER versus RTF of different approaches at various pruning beamwidths (150, 160, 180 and 200). Each data point on a curve corresponds to one particular pruning setting. The pruning beamwidth varies from the narrowest (the leftmost data point) to the widest (the rightmost data point).

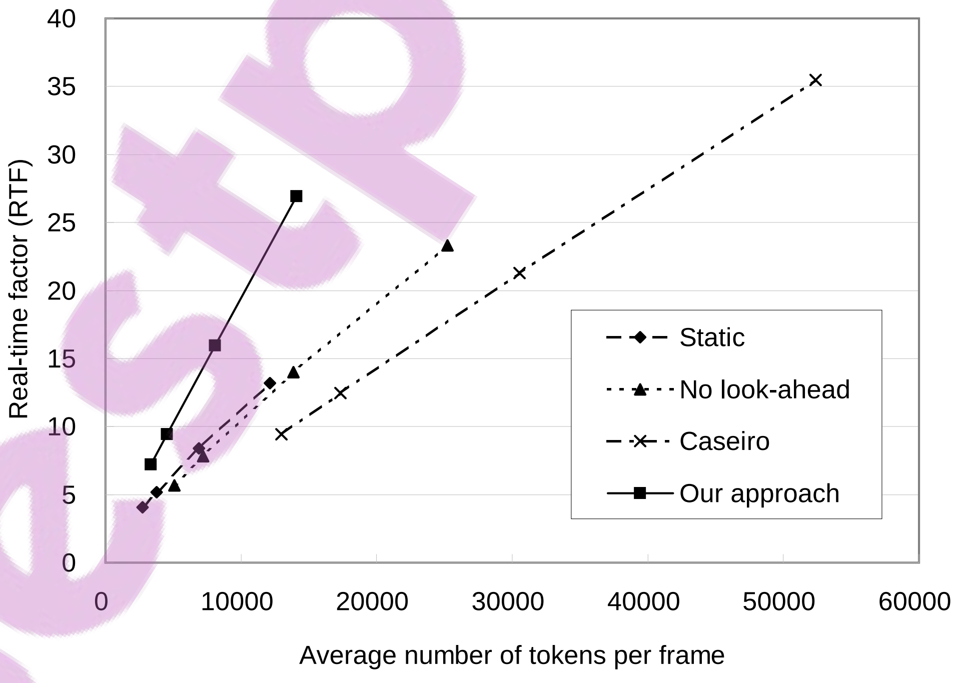

Figure 7.12: RTF versus Average number of tokens per frame of different approaches at various pruning beamwidths (150, 160, 180 and 200).

One of the major reasons to perform dynamic composition is the reduction in memory requirement. Table 7.1 compares the maximum memory usage (in MB) of our approach and the static approach. It shows a reduction of about 60% in memory usage. Since both the incremental approach and Caseiro's approach are also dynamic composition algorithms, they show a similar amount of reduction in memory requirement.

Beam width	Static	Dynamic (Our approach)	% reduction
150	1774	679	61.7
160	1775	697	60.7
180	1965	722	63.3
200	1966	762	61.2

Table 7.1: Maximum memory usage (in MB) during decoding

7.6 Summary

In static WFST composition, all the knowledge sources are integrated into one compact WFST. The static approach allows simple decoder design and faster decoding speed since the decoder does not need to combine various kinds of knowledge sources during decoding. However, once all the knowledge sources have been fully integrated, on-line modification of the knowledge sources becomes difficult. Furthermore, the size of the fully integrated transducer can be very large in large vocabulary tasks, resulting in great memory requirement during decoding.

One way of addressing these issues is to perform dynamic WFST composition during decoding. In this dynamic approach, the entire search space is factorized into two or more constituent WFSTs. The combination of these constituent WFSTs is carried out during decoding.

In this chapter, we have proposed a generalized dynamic WFST composition algorithm, which avoids the creation of non-coaccessible transitions, performs weight lookahead and does not impose any constraints to the topology of the WFSTs. Experimental results show that our weight lookahead approach gives better WER versus RTF characteristics than other dynamic composition approaches. Comparing with static composition, it shows a significant reduction in memory usage.

8 Conclusions

8.1 Hardware-software co-processing ASR system

The first objective of this thesis is to develop an embedded ASR system that is suitable for real-time applications. In this thesis, a hardware-software co-processing ASR system is proposed. As discussed in Section 1.2, this approach combines the advantages of the pure software-based and the pure hardware-based approaches. It provides rapid prototyping of applications, algorithmic acceleration, flexibility in design modification and system extensibility.

The co-processing system mainly consists of an embedded processor and a hardware accelerator which calculates the Gaussian mixture model (GMM) emission probabilities. The embedded processor is an Altera Nios II softcore processor running at 120MHz. The system is synthesized on an Altera Stratix II FPGA device. The target application is the 993-word Resource Management (RM1) task, which serves as a benchmark for a typical command-and-control application. There are 1200 utterances in the test set of the corpus. The performance of the proposed system is summarized as follows.

- The word accuracy rate is 93.33% with an average real-time factor of 0.62. Compared with the pure software-based baseline system, the word accuracy rate is the same and the decoding speed is about 3 times faster.
- As shown in Table 6.3, the word accuracy rate of our proposed system is within

the range of other reported systems. The average real-time factor of our proposed system is well below 1.00 and less (better) than those of other systems.

In comparison with [31, 62], which is also a hardware-software co-processing system, our proposed system possesses the following differences.

- Our proposed system uses a double-buffering technique to alleviate the memory bandwidth issue. The amount of acoustic parameters stored on the FPGA is kept to minimum. On the other hand, in [31, 62], a relative large percentage of acoustic parameters is required to be stored on the FPGA, which makes the design infeasible for larger vocabulary tasks.
- In our approach, we do not make any assumptions on the access pattern of the acoustic parameters. On the contrary, in order to observe a gain in timing performance, [31, 62] relies on storing the most frequently used acoustic parameters of the test data set on the FPGA. However, in practical applications, the access pattern of the acoustic parameters is related to the topic of the current conversation and varies in different situations. Therefore, the access pattern observed from the test data set can be very different from the practical situations. This makes the system less portable to different changing conditions.
- Our proposed GMM accelerator consists of more parallel pipelines (39 pipelines) than [31, 62] (1 pipeline), which further employs parallelism in GMM calculation.
- The real-time factor of the entire ASR algorithm in our proposed system is presented in detail. However, this timing measure is not clearly shown in [31, 62].

To further improve the timing performance, we propose an adaptive beam pruning algorithm, which applies tighter pruning when the number of active hypotheses exceeds a certain threshold. The performance of the adaptive beam pruning algorithm is summarized as follows.

- For the same RM1 task, the average real-time factor further reduces to 0.54. The percentage of the test utterances, which have a real-time factor of less than 1.00, increases from 94.08% to 99.75%. The word accuracy rate is 93.16%.
- The adaptive beam pruning algorithm only requires to count the number of active tokens and accordingly adjust the beamwidth. In terms of implementation, it is simpler than histogram pruning, where an insertion sort is performed for each of the active tokens in order to keep a sorted list of the tokens' scores.

In this part of the work, we also propose a framework for converting data formats from floating-point to fixed-point. The characteristics and the performance of the proposed framework are summarized as follows.

- For the RM1 task, the word accuracy rates of the floating-point and the fixed-point implementations are 93.09% and 93.33% respectively. It can be seen that the fixed-point system does not degrade the performance. In fact, for this RM1 task, the word accuracy rate of the fixed-point system is slightly higher than that of the floating-point system.
- The fixed-point system is about 9 times faster than the floating-point system on the Nios II platform.
- The proposed framework is a unified framework covering the conversion of all the data types within the ASR algorithm. On the contrary, previous studies focus on only a certain part of the ASR algorithm. For example, [22] describes the format conversion within MFCC feature extraction only, whereas [28, 30] focus on quantization within emission probability calculation.
- The proposed framework is tested on the 993-word RM1 task, while previous studies, such as [22, 30], are tested on small vocabulary tasks only (about 10 to 120 words).

8.2 Dynamic composition of WFST

The second objective of the thesis is to develop a dynamic composition algorithm of weighted finite state transducers (WFST). Specifically, the entire search space is factorized into two constituent WFSTs, which are then combined during decoding. Since modification of the constituent WFSTs is much easier than that of the fully-integrated WFST, dynamic composition allows on-line adaptation of the knowledge sources, as opposed to static composition. Another reason for dynamic composition is that the fully-integrated WFST in the static approach can be prohibitively huge if the knowledge sources are large and complex, which results in huge memory requirement during decoding.

The proposed algorithm is a generalized dynamic WFST algorithm which has the following characteristics.

- It does not impose any constraints on the topology of the constituent WFSTs. On the other hand, in [14], the topology of the lexicon WFST, $\tilde{L}$, is constrained.
- Symbol and weight lookahead is performed in all parts of the constituent WFSTs. In other studies, lookahead is only performed in certain parts of the constituent WFSTs [14], or it is not performed at all [20, 61].

- The proposed algorithm avoids the creation of the non-coaccessible transitions or so called the "dead-end" transitions.
- In the Wall Street Journal (WSJ1) 20k-word task tested on the desktop environment, the proposed algorithm has a better word accuracy rate versus real-time factor characteristics when compared with other studies [14, 20, 61]. In other words, at the same level of word accuracy rate, the real-time factor of our proposed algorithm is lower with a fewer number of active tokens.
- In comparison with the static approach, the proposed algorithm requires less amount of memory with an increase of decoding time. While the dynamic composition approach allows on-line adaptation of the knowledge sources, the static approach creates an optimized fully-integrated search space that permits simple decoder design and efficient decoding. It shows a trade-off between adaptability of knowledge sources and decoding speed.

8.3 Future work

In this thesis, an ASR system architecture consisting of an embedded processor and a hardware accelerator is proposed. The hardware accelerator serves as a co-processor which main purpose is to reduce the decoding time in order to meet the timing constraints. While decoding speed is an important requirement of an ASR system, other issues, such as resource usage and power consumption, also need to be addressed. In our implementation, the FPGA device is large enough to accommodate the entire hardware-software co-processing system. However, it does not stop designers from synthesizing the system on a smaller device in order to save production cost and power consumption. The system architecture should be flexible enough so that designers can quickly modify the architecture to suit different timing, resource and power requirements.

With this purpose in mind, one of the possible works is to develop a mechanism which allows easy customization of the hardware accelerator. One possible solution is to write a program or a script that accepts a set of specification as the input and automatically generates the hardware description of the accelerator as the output. For example, in our implementation of the hardware accelerator as illustrated in Figure 6.2, designers can decide the number of parallel pipelines in the datapath. The generator program can then create the hardware description of the hardware accelerator accordingly. This approach makes the hardware accelerator as a portable and flexible IP core which can be implemented in different target technology with different amount of resources.

In this thesis, a dynamic WFST composition algorithm has been proposed. Another possible future direction is to utilize the dynamic composition algorithm and develop an adaptable embedded ASR system. The system can include an adaptive module which is capable of modifying the underlying knowledge sources, for example, language model probabilities. In addition, the system can also adapt to changing knowledge sources, such as dialogue states and dynamic vocabulary. Furthermore, the system can accept new grammar rules and vocabulary introduced by users, which allows user involvement in the interface design and enhances user-friendliness.

A
Mel filter bank

The mel filter bank consists of a series of overlapping triangular bandpass filters. The filters are spaced uniformly on a non-linear frequency scale called the mel scale [59]. Figure A.1 illustrates the frequency response of the mel filter bank comprising M bandpass filters. The frequency scale shown in the figure is linear.

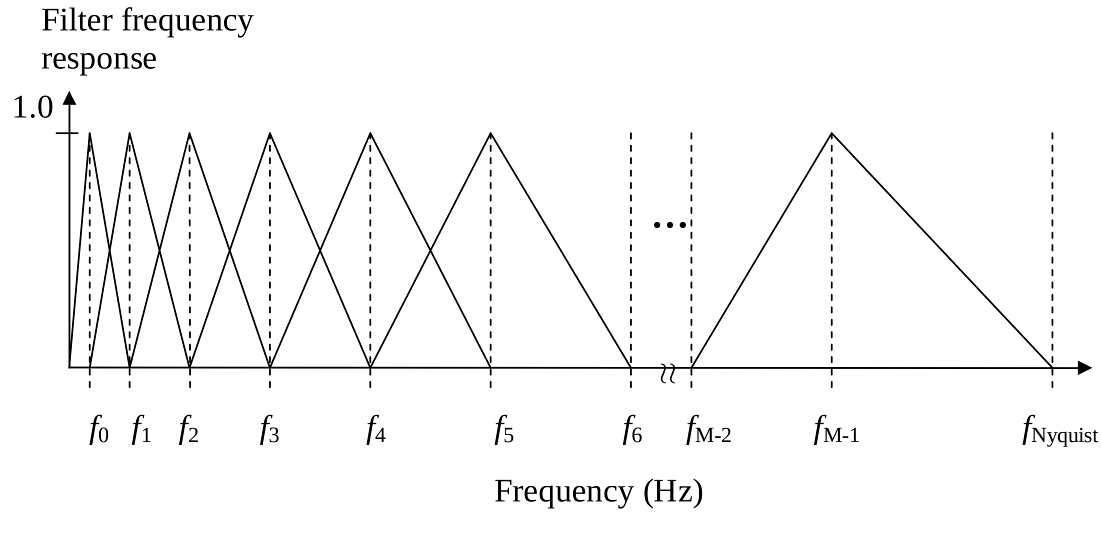

Figure A.1: Mel filter bank consisting of M triangular bandpass filters. f_m is the centre frequency of the m^{th} mel filter. $f_{Nyquist}$ is the Nyquist frequency.

The relationship between the mel scale and the linear frequency scale can be expressed mathematically by the following equation.

$$f_m^{(mel)} = 1127 \log \left(1 + \frac{f_m}{700}\right) \tag{A.1}$$

where $f_m^{(mel)}$ and f_m are the centre frequencies of the m^{th} filter in the mel scale and in the linear scale respectively. To position the bandpass filters on the mel scale, their centre frequencies are determined as follows.

$$f_m^{(mel)} = \frac{f_{Nyquist}^{(mel)}(m+1)}{M+1} \quad 0 \leq m < M \tag{A.2}$$

which indicates that the mel filters are spaced uniformly on the mel scale.

In Section 4.1, it shows that the output of the m^{th} mel filter, X_m, can be expressed by the following equation.

$$X_m = \sum_{k=0}^{\frac{K}{2}-1} F_{m,k} Y_k \quad 0 \leq m < M \tag{A.3}$$

where $F_{m,k}$ is the frequency response of the m^{th} filter and Y_k is the speech power spectrum at frequency index k. K is the number of FFT points. It can be seen that X_m reaches the maximum when Y_k is at the maximum. Hence, the equation can be rewritten as follows.

$$X_m^{(max)} = Y^{(max)} \sum_{k=0}^{\frac{K}{2}-1} F_{m,k} \quad 0 \leq m < M \tag{A.4}$$

where $Y^{(max)}$ is the maximum allowable value of Y_k. In addition, $X_m^{(max)}$ is the maximum value amongst all the possible m when $\sum_{k=0}^{\frac{K}{2}-1} F_{m,k}$ is the greatest. In other words, the mel filter which is the most widely spanned in the frequency domain gives the maximum value of $X_m^{(max)}$ amongst all the possible m. This filter is essentially the one with the highest centre frequency (the rightmost filter in Figure A.1).

In order to avoid potential overflows, it is necessary to find the maximum of $\sum_{k=0}^{\frac{K}{2}-1} F_{m,k}$. A computer program is written to calculate this value. As mentioned before, the filter with the highest centre frequency (f_{M-1}) is chosen for calculation. In our fixed-point implementation, there are 26 mel filters ($M = 26$). The Nyquist frequency is 8kHz ($f_{Nyquist} = 8000$). The number of FFT points is 512 ($K = 512$).

The results show that the maximum value of $\sum_{k=0}^{\frac{K}{2}-1} F_{m,k}$ is 23.68. It suggests that $X_m^{(max)}$ is 23.68 times the value of $Y^{(max)}$. Hence, as mentioned in Section 4.1.2, the least significant 5 bits ($2^4 < 23.68 < 2^5$) of $F_{m,k}Y_k$ is truncated before summation in order to prevent possible overflows.

B
DCT and Liftering

Mel frequency cepstral coefficients (MFCC) are calculated by performing discrete cosine transform (DCT) and liftering on the log mel filter outputs. The following equations demonstrate the DCT and liftering operations.

$$o^{(d)} = \sum_{m=0}^{M-1} \log(X_m) w_{m,d} \qquad 0 \leq d < D_{static} \tag{B.1}$$

$$w_{m,d} = l_d \sqrt{\frac{2}{M}} \cos\left[\frac{\pi d}{M}(m + 0.5)\right] \qquad 0 \leq d < D_{static}; 0 \leq m < M \tag{B.2}$$

$$l_d = 1 + \frac{L}{2} \sin\left(\frac{\pi d}{L}\right) \qquad 0 \leq d < D_{static} \tag{B.3}$$

In (B.1), $o^{(d)}$ is the d^{th} MFCC coefficient. The total number of MFCC coefficients is D_{static}. The log output of the m^{th} mel filter is denoted by $\log(X_m)$. There are M mel filters in the filter bank. The weighting coefficient, $w_{m,d}$ as defined in (B.2), combines the effect of DCT and liftering. In (B.3), l_d is the liftering coefficient and L is a constant.

In order to determine the precision format of $o^{(d)}$, its dynamic range needs to be found. In (B.1), we assume that $\log(X_m)$ is either $+\mathcal{X}$ or $-\mathcal{X}$, where $\mathcal{X}$ is the maximum allowable value of $\log(X_m)$. Hence, the dynamic range of the static MFCC coefficient can be expressed by the following equation.

$$-\mathcal{X}\sum_{m=0}^{M-1}|w_{m,d}| \leq o^{(d)} \leq \mathcal{X}\sum_{m=0}^{M-1}|w_{m,d}| \quad 0 \leq d < D_{static} \tag{B.4}$$

According to the above equation, we need to evaluate the term $\sum_{m=0}^{M-1}|w_{m,d}|$ in order to determine the dynamic range of the static coefficient. A computer program is written to calculate this term for $0 \leq d < D_{static}$. The other parameters are: $D_{static} = 13$, $M = 26$ and $L = 22$. The results are shown in the following table.

d	$\sum_{m=0}^{M-1}\|w_{m,d}\|$
0	7.21
1	11.78
2	18.73
3	25.58
4	31.97
5	37.68
6	42.55
7	47.10
8	50.65
9	53.08
10	54.31
11	55.12
12	54.71

Table B.1: Values of $\sum_{m=0}^{M-1}|w_{m,d}|$ for $0 \leq d < D_{static}$. The other parameters include $D_{static} = 13$, $M = 26$ and $L = 22$.

In the table, the maximum value is 55.12, which can increase the number of integer bits by at most 6 bits. As discussed in Section 4.1.2, $\log(X_m)$ and $w_{m,d}$ are in Q5.10 and Q2.13 formats respectively. In (B.1), the multiplication by $w_{m,d}$ increases the number of fractional bits of $\log(X_m)$ by 13 and the summation increases the number of integer bits by at most 6 bits. Thus, the format of $o^{(d)}$ becomes Q11.23. In order to keep it in 32-bit format, the least significant 3 bits are truncated, resulting in the fixed-point format of Q11.20.

References

[1] F. Alleva, X. Huang, and M. Hwang. Improvements on the pronunciation prefix tree search organization. In *Proceedings of ICASSP*, 1996.

[2] Altera Corporation. *Nios Development Board Reference Manual, Stratix II Edition*, 2005.

[3] Altera Corporation. *Nios II Processor Reference Handbook*, 2006.

[4] Altera Corporation. *Avalon Memory-mapped Interface Specification*, 3.3 edition, 2007.

[5] B. Atal. Effectiveness of linear prediction characteristics of the speech wave for automatic speaker identification and verification. *J. Acoust. Soc. Amer.*, 55(6):1304–1312, 1974.

[6] X. Aubert. An overview of decoding techniques for large vocabulary continuous speech recognition. *Computer Speech and Language*, 16:89–114, 2002.

[7] L. Baum, T. Petrie, G. Soules, and N. Weiss. A maximization technique ocurring in the statistical analysis of probabilistic functions of markov chains. *Ann. of Math. Stat.*, 41(1):164–171, 1970.

[8] J. Bilmes. A gentle tutorial of the EM algorithm and its application to parameter estimation for gaussian mixture and hidden Markov models. TR-97-021, International Computer Science Institute, 1998.

[9] E. Bocchieri and D. Blewett. A decoder for LVCSR based on fixed-point arithmetic. In *Proceedings of ICASSP*, 2006.

[10] H. Bourlard and N. Morgan. *Connectionist Speech Recognition: A Hybrid Approach.* Kluwer Academic Publishers, Boston, 1994.

[11] D. Caseiro and I. Trancoso. On integrating the lexicon with the language model. In *Proceedings EUROSPEECH'2001*, 2001.

[12] D. Caseiro and I. Trancoso. Transducer composition for "on-the-fly" lexicon and language model integration. In *Proceedings ASRU'2001*, 2001.

[13] D. Caseiro and I. Trancoso. A tail-sharing WFST composition algorithm for large vocabulary speech recognition. In *Proc. ICASSP*, 2003.

[14] D. Caseiro and I. Trancoso. A specialized on-the-fly algorithm for lexicon and language model composition. *IEEE Tran. on Audio, Speech and Language Processing*, 14(4), 2006.

[15] C. Chen, J. Bilmes, and K. Kirchhoff. Low-resource noise-robust feature post-processing on Aurora 2.0. In *Proceedings of ICSLP*, 2002.

[16] O. Cheng, J. Dines, and M. Magimai-Doss. A generalized dynamic composition algorithm of weighted finite state transducers for large vocabulary speech recognition. In *Proceedings of ICASSP*, 2007.

[17] S. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 28(4):357–366, 1980.

[18] A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. Roy. Statist. Soc. Ser. B*, 39:1–38, 1977.

[19] P. Dixon, D. Caseiro, T. Oonishi, and S. Furui. The TITECH large vocabulary WFST speech recognition system. In *Proceedings of ASRU*, 2007.

[20] H. Dolfing and I. Hetherington. Incremental langauge models for speech recognition using finite-state transducers. In *Proc. ASRU*, 2001.

[21] S. Furui. Speaker independent isolated word recognition using dynamic features of speech spectrum. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 34(1):52–59, 1986.

[22] Y. Gong and Y. Kao. Implementing a high accuracy speaker-independent continuous speech recognizer on a fixed-point DSP. In *Proceedings of ICASSP*, 2000.

[23] H. Hermansky. Perceptual linear predictive (PLP) analysis for speech. *J. Acoust. Soc. Amer.*, 87(4):1738–1752, 1990.

[24] L. Hetherington. The MIT FST Toolkit. Available on the Internet: http://people.csail.mit.edu/ilh//fst/, 2005.

[25] T. Hori and A. Nakamura. Generalized fast on-the-fly composition algorithm for WFST-based speech recognition. In *Interspeech*, 2005.

[26] D. Huggins-Daines, M. Kumar, A. Chan, A. Black, M. Ravishankar, and A. Rudnicky. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *Proceedings of ICASSP*, 2006.

[27] F. Jelinek. A fast sequential decoding algorithm using a stack. *IBM J. Research and Dev.*, 13, 1969.

[28] S. Kanthak, K. Schütz, and H. Ney. Using SIMD instructions for fast likelihood calculation in LVCSR. In *Proceedings of ICASSP*, 2000.

[29] S. Katz. Estimation of probabilities from sparse data for language model component of a speech recognizer. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 35(3):400–401, 1987.

[30] J. Leppänen and I. Kiss. Comparison of low footprint acoustic modeling techniques for embedded ASR systems. In *Interspeech*, 2005.

[31] H. Lim, K. You, and W. Sung. Design and implementation of speech recognition on a softcore based FPGA. In *Proceedings of ICASSP*, 2006.

[32] E. Lin, K. Yu, R. Rutenbar, and T. Chen. Moving speech recognition from software to silicon: the In Silico Vox Project. In *Interspeech*, 2006.

[33] E. Lin, K. Yu, R. Rutenbar, and T. Chen. A 1000-word vocabulary, speaker-independent, continuous live-mode speech recognizer implemented in a single FPGA. In *International Symposium on Field-Programmable Gate Arrays*, 2007.

[34] J. Makhoul. Linear prediction: A tutorial review. *Proc. IEEE*, 63:561–580, 1975.

[35] M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

[36] M. Mohri. Weighted finite-state transducer algorithms: An overview. *Formal Languages and Applications*, 148:551–564, 2004.

[37] M. Mohri, F. Pereira, and M. Riley. AT&T FSM Library - Finite-State Machine Library. Available on the Internet: http://www.research.att.com/~fsmtools/fsm/, 1997.

[38] M. Mohri, F. Pereira, and M. Riley. Weighted finite state transducers in speech recognition. In *ISCA ITRW Automatic Speech Recognition: Challenges for the Millennium*, 2000.

[39] M. Mohri and M. Riley. A weight pushing algorithm for large vocabulary speech recognition. In *Proc. Eurospeech*, 2001.

[40] M. Mohri, M. Riley, D. Hindle, A. Ljolje, and F. Pereira. Full expansion of context-dependent networks in large vocabulary speech recognition. In *Proceedings of ICASSP*, 1998.

[41] D. Moore. The Juicer LVCSR Decoder - user manual for Juicer version 0.5.0. IDIAP-COM 03, IDIAP, 2006.

[42] D. Moore, J. Dines, M. Magimai Doss, J. Vepa, O. Cheng, and T. Hain. Juicer: A weighted finite-state transducer speech decoder. In *MLMI'06*, 2006.

[43] H. Ney, R. Haeb-Umbach, B. Tran, and M. Oerder. Improvements in beam search for 10000-word continuous speech recognition. In *Proceedings of ICASSP*, 1992.

[44] H. Ney, D. Mergel, A. Noll, and A. Paeseler. A data driven organization of the dynamic programming beam search for continuous speech recognition. In *Proceedings of ICASSP*, 1987.

[45] M. Novak. Towards large vocabulary ASR on embedded platforms. In *Interspeech*, 2004.

[46] S. Ortmanns, H. Ney, and A. Eiden. Language-model look-ahead for large vocabulary speech recognition. In *Proceedings of ICSLP*, 1996.

[47] D. Paul. An efficient A* decoder algorithm for continuous speech recognition with a stochastic language model. In *Proceedings of ICASSP*, 1992.

[48] D. Paul and J. Baker. The design for the Wall Street Journal-based CSR corpus. In *Proc. ICSLP*, 1992.

[49] L. Paulson. Speech recognition moves from software to hardware. *IEEE Computer*, 39(11):15–18, 2006.

[50] F. Pereira and M. Riley. Speech recognition by composition of weighted finite automata. *Finite-State Language Processing*, pages 431–453, 1997.

[51] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical recipes in C: the art of scientific computing.* Cambridge University Press, Cambridge, 1992.

[52] P. Price, W. Fisher, J. Bernstein, and D. Pallett. The DARPA 1000-word resource management database for continuous speech recognition. In *Proceedings of ICASSP*, 1988.

[53] L. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. IEEE*, 77(2):257–286, 1989.

[54] L. Rabiner and B. Juang. *Fundamentals of speech recognition.* Prentice Hall, Inc., 1993.

[55] L. Rabiner and R. Schafer. *Digital Processing of Speech Signals.* Prentice Hall, Inc., 1978.

[56] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 26(1):43–49, 1978.

[57] J. Schuster, K. Gupta, R. Hoare, and A. Jones. Speech Silicon: An FPGA architecture for real-time, Hidden Markov Model based speech recognition. In *EURASIP Journal on Embedded Systems*, 2006.

[58] V. Steinbiss, B. Tran, and H. Ney. Improvements in beam search. In *Proceedings of ICSLP*, 1994.

[59] S. Stevens, J. Volkmann, and E. Newman. A scale for the measurement of the psychological magnitude pitch. *J. Acoust. Soc. Amer.*, 8:185–190, 1937.

[60] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Trans. on Information Theory*, 13(2):260–269, 1967.

[61] D. Willett and S. Katagiri. Recent advances in efficient decoding combining on-line transducer composition and smoothed language model incorporation. In *Proc. ICASSP*, 2002.

[62] K. You, H. Lim, and W. Sung. Architectural design and implementation of an FPGA softcore based speech recognition system. In *The 6th International Workshop on System on Chip for Real Time Applications*, 2006.

[63] S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. Liu, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchevnt, and P. Woodland. *The HTK Book (for HTK Version 3.4).* Cambridge University Engineering Department, 2006.

[64] S. Young, N. Russell, and J. Thornton. Token passing: a simple conceptual model for connected speech recognition systems. CUED/F-INFENG/TR38, Cambridge University Engineering Dept, 1989.