REAL-TIME SPEAKER-INDEPENDENT LARGE VOCABULARY CONTINUOUS SPEECH RECOGNITION

A Dissertation

Presented to

The Faculty of the Graduate School

University of Missouri – Columbia

In Partial Fulfillment

Of the Requirement for the Degree

Philosophy Doctorate of Computer Science

---

by

XIAOLONG LI

Dr. Yunxin Zhao, Dissertation Supervisor

December 2005

Acknowledgements

Firstly, I want to thank my parents and my wife for their endless love. Thank my parents for giving me life and continuous support for my academic pursuit. Thank my wife for building up a happy family for me and her tolerance with my shortcomings. Their love makes me strong and energetic.

Secondly, I want to thank my advisor, Dr. Yunxin Zhao, for her excellent guidance on my research projects during my PhD study in the University of Missouri-Columbia. She always inspires me with insightful ideas. And she always encourages me to face challenges with endurance and hard work. Without her help, I would have never achieved so much in my research.

Thirdly, I would like to thank Dr. Zhuang, Dr. Shi, Dr. Ho, and Dr. Uhlmann for their help in my academic study as my committee members. Thank them for reviewing my dissertation and giving me valuable advices.

Fourthly, I would like to thank all my former and current colleagues in the Laboratory of Spoken Language and Information Processing, including Xiaodong He, Xiao Zhang, Kaile Li, Rusheng Hu, Rong Hu, Jian Xue, Xiaojia Zhang and Lili Che. Thank them for all the great time we have together in both research and life.

Lastly, I would like to thank my friend George H. Wagner, an emeritus professor of Soil Microbiology/Biochemistry department at the University of Missouri, for his help in

English.

Table of Contents

List of Tables

List of Figures

Abstract

In this dissertation, a real-time decoding engine for speaker-independent large vocabulary continuous speech recognition (LVCSR) is presented. An overview is first given covering the state-of-the-art decoding algorithms for LVCSR. Since accuracy, speed, and memory cost are three indispensable and correlated performance measurements for a practical continuous speech recognition system, all three aspects are carefully considered, with the main innovations in fast and memory-efficient decoding algorithms.

For accuracy, crossword triphone based Hidden Markov Model (HMM) is used in the developed system, which has been proved to generate significantly higher accuracy than within-word triphone HMM. With the use of crossword triphone model, the search space dramatically increases compared with the system using within-word triphone model. Crossword Language Model lookahead and fan-out arc tying are used to make the search space as compact as possible. Five heuristic pruning methods and two lookahead techniques are also exploited to reduce the search space with little or no loss of accuracy.

For the aspects of speed and memory cost, a novel algorithm, Order-Preserving Language Model Context Pre-computing (OPCP) is proposed for fast Language Model (LM) lookup, resulting in significant improvement in both overall decoding time and memory space without any decrease of recognition accuracy. OPCP is a novel integration of two previously proposed methods: Minimum Perfect Hashing (MPH) and Language Model Context Pre-computing (LMCP). By reducing hashing operations through order-

preserving access of LM scores, OPCP cuts down LM lookup time effectively. In the meantime, OPCP significantly reduces memory cost because of reduced size of hashing keys and the need for only last word index of each N-gram in LM storage. Experimental results are reported on two LVCSR tasks (Wall Street Journal 20K and Switchboard 33K) with three sizes of trigram LMs (small, medium, large). In comparison with MPH and LMCP methods, OPCP reduced LM lookup time from about 30~80% of total decoding time to about 8%~14%, without any loss of word accuracy. Except for the small LM, the total memory cost of OPCP for LM lookup and storage was about the same or less than the original N-gram LM storage, and was much less than the compared methods. The time and memory savings in LM lookup by using OPCP became more pronounced with the increase of LM size.

By using the OPCP method and other optimizations mentioned above, our one-pass LVCSR decoding engine, named TigerEngine, reached real-time speed in both tasks of Wall Street Journal 20K and Switchboard 33K, on the platform of a Dell workstation with one 3.2 GHz Xeon CPU.

Chapter 1

Motivation and Contributions

1.1 Introduction

Speech is one of the most common and useful means of communication for human beings. However, it has been a long-time dream for humans to use speech to communicate with a computer [Rabiner&Juang 93]. Compared with today's standard user interfaces such as keyboard, mouse, touch-screen, or joy-stick, speech owns overwhelming advantages in speed and interface friendliness. Even an illiterate or person with little knowledge about computer may use speech to operate one. Many disabled people may use a computer with the help of speech input in case they are unable to type in keyboard or click in mouse with their hands. For normal people or experienced people in computer technologies, they can utilize the speech-input ability of a computer to significantly speedup documentation writing, email sending, web searching and other operations with a computer. Another advantage of speech-input is that it can be used for many situations when the hands are

already used for important operations such as driving a car. Speech-enabled dialing and GPS navigation are two examples. With the development of Machine Translation techniques, another exciting application called Automatic Spoken Language Translation (or speech-to-speech translation [Lavie+ 97]) has emerged, which allows people from different countries all over the world to be able to freely communicate via speech without any professional translator.

Speech recognition is an automatic technology that allows a computer to understand human speech. Here "understanding" means to transfer the speech waveform into meaningful text, called Speech-To-Text (STT)[1]. In spite of its promising prospect in the future, the complexity of human's speech production and perception makes speech recognition a very difficult and complicated task and it is deeply involved with multiple disciplines including acoustics, phonetics, linguistics, anatomy, physiology, neuroscience, computer science, electrical engineering, artificial intelligence and signal processing [Huang+ 03].

My research aims at one of the most difficult tasks of speech recognition, i.e., speaker-independent (SI) large vocabulary continuous speech recognition (LVCSR), with the main goal on developing algorithms for real-time LVCSR decoding. Generally speaking, SI-LVCSR includes the following technical highlights:

- Large training corpus collected from hundreds or thousands of speakers, with the total

[1] For higher-level processing of speech such as understanding the meaning of a recognized sentence, it is usually called "speech understanding", which is different from "speech recognition".

length of speech data ranging from several hours to hundreds of or even thousands of hours [Evermann+ 05]; on the other hand, the recognition system will work for multiple speakers with little or no adaptation data.

- System input is continuous speech, which is the same as a human's common speaking style, and which is distinguished from isolated word recognition (such as command & control) or connected digital recognition (such as recognition of telephone number and credit card account).
- Vocabulary size is larger than 1,000. A typical vocabulary for an English LVCSR system is from 5,000 to 65,000.

1.2 Motivation

A practical LVCSR system needs both high accuracy and fast speed. At the same time, lower memory cost is also required for its deployment in affordable computer systems. Most of previous research work only focused on recognition accuracy while ignoring system speed and memory cost. A real-time speech recognition system means that the average decoding time of a sentence will not be longer than the time the speaker took to speak it. The importance of a real-time system lies in the following aspects. Firstly, long-time delay will seriously affect the friendliness of a speech recognition system. Nobody will use a dictation system that takes several days to recognize half an hour of speech. Also a telephone-based automatic weather retrieval system will not be accepted if

it needs 10 minutes to answer one query. Secondly, a lot of important applications of LVCSR need real-time responses such as automatic television/movie captioning, assistive system for hearing-impaired people, and spoken language based stock transaction systems. Thirdly, the challenges for LVCSR in a noisy environment and spontaneous speaking style have caused the algorithms to become more and more complex in acoustic modeling and language modeling, and so the system requires more and more CPU resources to reach high-accuracy, making fast and memory-efficient decoding more important.

Some may argue that speed is really not an issue as programs become faster with each new generation of computers (by Moore's Law). This is only one side of the fact, however. Firstly, on one hand, even the fastest general-purpose single-CPU computer of today (such as Dell's Xeon workstation with clock rate of 3.4G Hz) can not guarantee real-time speed for a LVCSR system without speed-oriented optimization; with the help of fast decoding algorithms, on the other hand, real-time LVCSR dictation has already emerged since 1990s in many commercial systems such as IBM's ViaVoice, Dragon's NaturallySpeaking. Many application fields and commercial values of LVCSR will be lost in just waiting for a faster computer. Secondly, LVCSR systems are becoming more complex at a higher rate than computers get faster, and a speech recognition system used as an input interface is not supposed to take up 100% of the available cycles. Thirdly, for those source-limited intelligence-computing devices such as

PDA and Smart Phone, the increase of CPU processing speed is far behind the pace of desktop CPU indicated by Moore's Law, but these devices represent one of the most promising fields for the applications of LVCSR.

1.3 Contributions

The main contributions of this dissertation include:

- A systematic study of the state-of-the-art decoding algorithms of LVCSR. Those algorithms include Viterbi, A*, One-Pass, Multiple-Pass, and some of their combinations.
- Development of a one-pass LVCSR decoding engine called TigerEngine based on time-synchronous Viterbi-beam algorithm, one of the most successful decoding algorithms. TigerEngine 1.0 has already achieved real-time speed and state-of-the-art accuracy in benchmark testing of Wall Street Journal task with a vocabulary size of 20,000.
- TigerEngine uses crossword triphone based Hidden Markov Model (HMM) as basic acoustic model, which has been proved much better than within-word triphone based HMM. With the use of the crossword triphone model, the search space dramatically increased compared with the within-word triphone model. Aiming at the fast performance of TigerEngine, several techniques are implemented to make the search space as compact as possible with little or no loss of recognition accuracy. Those

techniques include heuristic pruning and lookahead methods.

- A novel algorithm based on Minimum Perfect Hashing (MPH) is presented for speeding up one important component of the decoding engine -- Language Model (LM) lookup. The algorithm, called Order-Preserving Context Pre-computing (OPCP), has significant advantages in both faster speed and memory saving. It is worth noting that OPCP speeds up the decoding without any loss of recognition accuracy.

This dissertation is organized as follows. Chapter 2 presents an overview of current knowledge of LVCSR and decoding algorithms. Chapter 3 describes the system development of a crossword triphone HMM and trigram language model based one-pass LVCSR decoding system, TigerEngine. In Chapter 4, the main innovations on fast Language Model lookup, i.e., OPCP algorithm are presented, together with the experimental results on two benchmark tasks, i.e., Wall Street Journal 20K, and Switchboard 33K. In Chapter 5, some conclusions are made and future directions are outlined.

Chapter 2

Overview of LVCSR and Decoding Algorithms

2.1 Continuous Speech Recognition

2.1.1 Maximum A Posteriori (MAP) Decision

Till now the most successful method of speech recognition is based on statistical pattern recognition. In this method, the problem of speech recognition is formulated in Bayesian Maximum A Posteriori (MAP) decision as: ([Ney&Ortmanns 99])

$$\begin{aligned}\hat{\mathbf{W}} &= \arg\max_{\mathbf{W}} P(\mathbf{W}\,|\,\mathbf{O}) \\ &= \arg\max_{\mathbf{W}} \frac{P(\mathbf{O}\,|\,\mathbf{W})P(\mathbf{W})}{P(\mathbf{O})} \\ &= \arg\max_{\mathbf{W}} P(\mathbf{O}\,|\,\mathbf{W})P(\mathbf{W})\end{aligned} \tag{2.1}$$

where $\mathbf{O} = \mathbf{o}_1, \mathbf{o}_2, ..., \mathbf{o}_T$ is the sequence of acoustic vectors, $\mathbf{W} = w_1, w_2, \ldots, w_n$ is the sequence of words in an utterance intended by the speaker which generates the acoustic vectors $\mathbf{O}$. $P(\mathbf{O}\,|\,\mathbf{W})$ is the probability that the speaker produces the acoustic data $\mathbf{O}$ if

W is the intended word sequence. $P(\mathbf{W})$ is a priori probability for the word sequence hypothesis, which is independent of observation vector **O** and is determined by language model. $P(\mathbf{O})$ is the probability of the observation vectors, which is independent of all word sequence hypotheses, so that it can be ignored in the last line of formula (2.1).

According to (2.1), the problem of statistical speech recognition can be decomposed into those following sub-problems:

- **Acoustic Model**, i.e., $P(\mathbf{O}|\mathbf{W})$, which only considers how likely the observation vectors are output by a word sequence hypothesis and its sub-components such as phonemes. The most commonly used acoustic model is Hidden Markov Model (HMM), which will be discussed in Section 2.1.2.
- **Language Model**, i.e., $P(\mathbf{W})$, which only considers a priori possibility of word sequence in a sentence. The most commonly used language model is statistical N-gram[2], which will be discussed in Section 2.1.3.
- **Pronunciation Model**. A model to decompose the word-level hypothesis into sub-word hypotheses, usually phonemes. A knowledge-based dictionary is usually used, which will be discussed in Section 2.1.4[3].
- **Speech pre-processing**. A procedure to convert the original waveform of speech into a type of presentation which can be easily processed by a computer. Common

[2] Another important Language Model is Context Free Grammar (CFG), which is mainly used for command & control applications. Please refer to [Huang+01] for the details.

[3] There is a statistical technology to transform a word into its phoneme list on-the-fly, i.e., Letter-To-Sound (LTS), which is usually used in Text-to-Speech (TTS). Also a statistical pronunciation model can be used for automatic selection of a pronunciation in spontaneous speech recognition. Also refer to [Huang+01].

pre-processing includes following steps [Huang+01]:

1. An analog-to-digital conversion to sample speech signal at discrete time intervals and quantize the samples into a finite set of values.
2. A fixed-length window based frame segmentation (e.g., 25-ms window with 15-ms overlap). All the following processing steps are based on a frame-by-frame fashion, which is also called **Short-Time Analysis**.
3. A high-pass filtering based pre-emphasis to counteract an intrinsic 6 dB/octave lowering in high-frequency components of speech signal.
4. A FFT-based transformation to convert the time-domain presentation of speech signal into the spectral-domain presentation. The main purposes of this processing exist in two folds: to save the memory by compression and to extract more discriminant information in spectral domain.
5. A filter-bank (critical band) and Mel-frequency (logarithmic scale) based processing which is similar to the hearing processing in human's ear.

Pre-processing may result in different types of concise representations of speech signal in a frame by frame fashion, such as Mel-Frequency Cepstral Coefficient (MFCC) [Davis&Mermelsten 80], Linear Prediction Coefficient (LPC) [Atal&Schroeder 67], Perceptual Linear Prediction (PLP) [Hermansky 90], LPC Cepstrum (LPCC) [Rabiner&Juang 93], etc. Those different vectors are called **feature vectors**. The dimension of feature vectors is usually 11 ~ 16 per frame in acoustic modeling, which is

much more compact than original time-domain presentation (e.g. 400 samples in a frame with 25-ms window and 16K Hz sampling rate). The above steps are also called **front-end processing** or **feature extraction**.

- **Sequential Decoding**. A problem of finding the word sequence **W** for a feature sequence **O** that maximizes the posteriori probability P(**W**|**O**). **Dynamic Programming** [Ney&Ortmanns 99] is the most successful solution for this problem, and it is the main research topic of this dissertation. Other useful solutions include A* algorithm, multi-pass based decoding, and so on. Decoding algorithms will be discussed in Section 2.1.5.

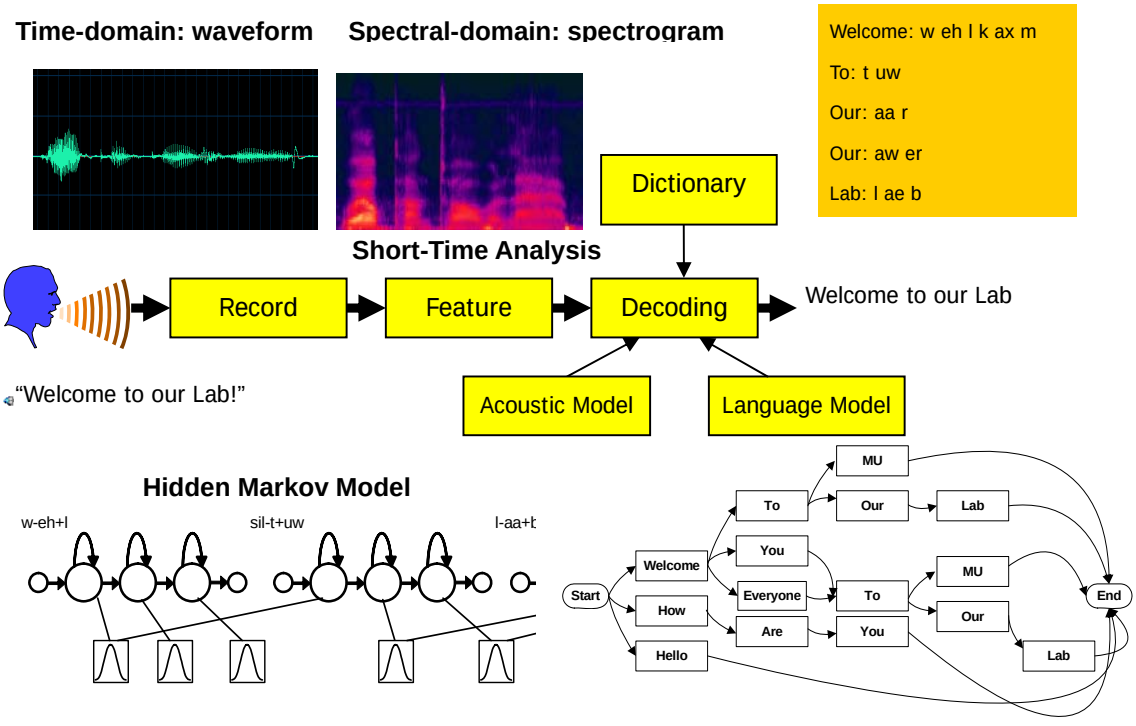

Figure 2.1. Components of a continuous speech recognition system. It shows the procedure of decoding an example sentence as "Welcome to our Lab!".

Figure 2.1 illustrates all the components of a continuous speech recognition system and the procedure of recognizing an example sentence as “Welcome to Our Lab”. Figure 2.2 illustrates the sequential decoding procedure for the same sentence by integrating the knowledge sources of acoustic model, dictionary, and language model.

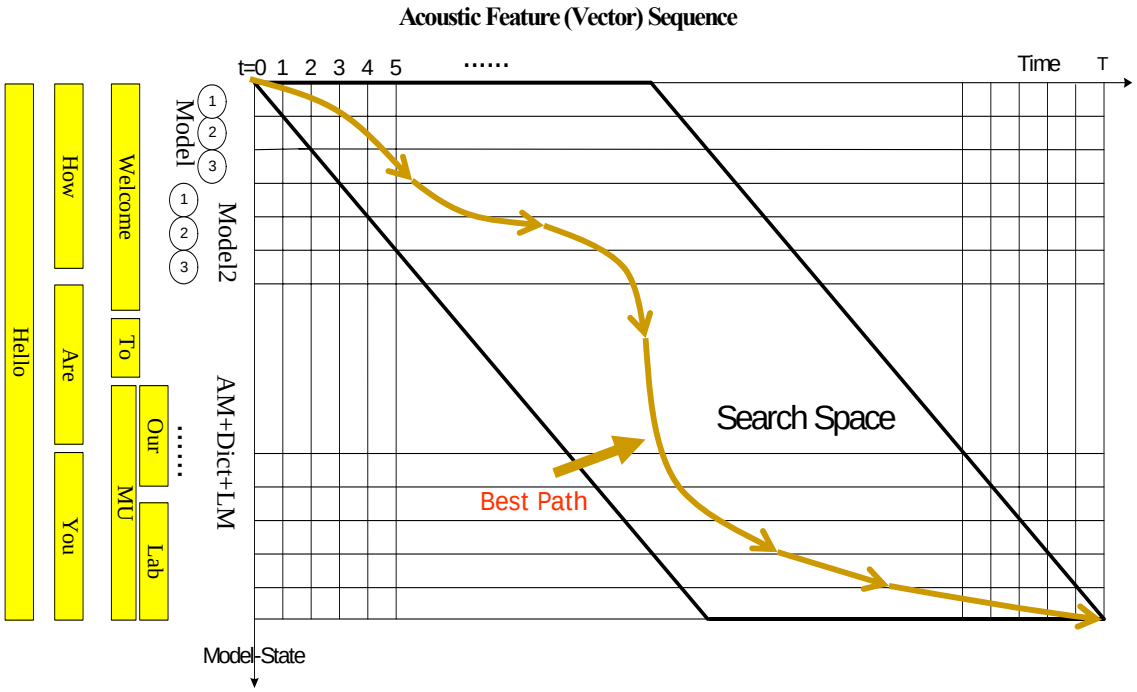

Figure 2.2. The sequential decoding of continuous speech recognition.

2.1.2 Acoustic Model

Acoustic Model is used to model the acoustic-phonetic characteristics of speech signal, where speech signal is considered as the output of a stochastic process, and the generative

and distribution properties of this stochastic process are described. In most commonly used Hidden Markov Model (HMM) [Rabiner 89], speech signal is generated by a Markov chains of hidden states, and each state is associated with a stationary process which is a Gaussian distribution or **Gaussian Mixtures** with weighted multiple Gaussians. The transitions between different states represent the non-stationary time-variation in speech signal. Figure 2.3 illustrates a typical 5-state HMM structure used in acoustic modeling of phoneme units for speech recognition [Young+ 00]. This HMM includes 3 emitting states and 2 non-emitting states. Three emitting states (indexed as 1, 2, 3) can generate speech observations with a Gaussian distribution (or Gaussian Mixtures) and each state is permitted to transit to itself as well as its next state. The two non-emitting states (indexed as 0 and 4) are merely used to connect with neighboring HMMs and have no output. This left-to-right topology of HMM is used to describe the temporal characteristics of speech signal, that is, the current state is only dependent on itself and its previous states, and not dependent on future states.

The main advantage of HMM lies in its double-randomness property: the state-transition probability ($a_{ij}, 0 \le i, j \le N-1$, N is the number of states) represents the temporal characteristics, and the state-output probability ($P(\mathbf{O}|S_i)$, i = 1, 2, 3) represents the short-time variants of spectral characteristics which account for distinctions among different phonemes. The hidden states of HMM correspond to different stages in producing a phoneme, i.e., onset, acoustically stable regions, and offset.

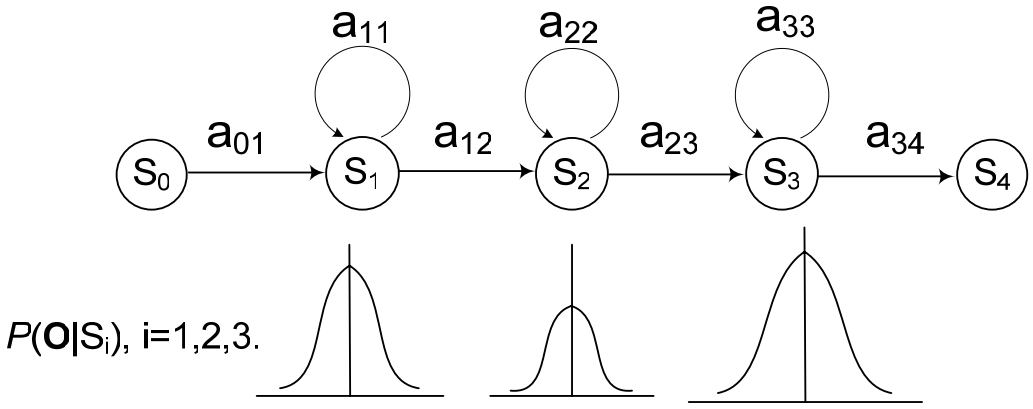

Figure 2.3. An example of HMM for a phoneme model.

In general, the output probability of a HMM state can be modeled by Gaussian mixture density as below:

$$P(\mathbf{O} \mid S) = \sum_{m=1}^{M} \frac{C_m}{(2\pi)^{d/2} \mid \mathbf{\Sigma}_m \mid^{1/2}} \exp\{-\frac{1}{2}(\mathbf{O} - \boldsymbol{\mu}_m)^T \mathbf{\Sigma}_m^{-1} (\mathbf{O} - \boldsymbol{\mu}_m)\} \tag{2.2}$$

where M is the number of Gaussians, $\boldsymbol{\mu}_m$ and $\mathbf{\Sigma}_m$ are the mean vector and covariance matrix for the m-th Gaussian component, C_m is the weight of the m-th Gaussian component with the constraints $C_m > 0, \sum_{m=1}^{M} C_m = 1, (m = 1,...., M)$.

There are two other advantages making HMM the most successful tools for speech recognition since its application in 1970s [Baker 75; Bahl+ 87], which are described as below:

Firstly, HMM has a set of compact mathematical presentations with fully developed algorithms for its parameter optimization and state decoding. The training algorithms

are based on well-known Expectation-Maximization (EM) algorithm in machine learning field and Maximum Likelihood (ML) principle [Dempster+ 77]. Two different algorithms called Baum-Welch [Baum 72] and Segmental K-means [Juang&Rabiner 90] are very powerful to handle large training corpus of LVCSR. On the other hand, the decoding algorithm, based on Dynamic Programming and A* algorithms, are also relatively simple and easy for implementation.

Secondly, the concept of HMM can be easily extended beyond the phoneme level. Word can also be regarded as a left-to-right HMM with each state being its component phonemes. If a stochastic pronunciation model is also considered with multi-pronunciations per word, then the state-transition probabilities of this word-level HMM are simply the probabilities of different pronunciations for a word. If a fixed pronunciation dictionary is used, then the state-transition probability will be equal to one for a_{ij} in word-level HMM. Similar extension works for the sentence level, where a whole sentence can be regarded as a composite HMM with the state-transition probability simply being word transition probability or Language Model probability. Figure 2.4 illustrates the topologies of word-level and sentence-level HMMs. Therefore, the decoding of a sentence can be regarded as a sequential decoding problem for this composite HMM in the same paradigm as phoneme-level HMM. This brings significant benefits for the implementation of LVCSR decoding algorithms based on HMM.

There are alternative approaches other than HMM emerged in recent years for

acoustic modeling, such as Artificial Neural Networks (ANN) [Morris+ 00], Segment Model [Ostendorf&Roukos 89], Graphical Model [Bilmes&Zweig 02] and Hidden Dynamic Model (HDM) [Deng&O'Shaughnessy 03]. Some of them are generalized cases of HMM such as Segment Model and Graphical Model, but others are critical departures from conventional HMM concepts. Some of the new models have shown better properties than HMM in noisy-robustness, better representation of context-dependency, and stronger representation of speech signal dynamics. However, till now none of them has beat HMM in the aspects of simple mathematical representation and ease of implementation for both training and decoding algorithms, especially for LVCSR task.

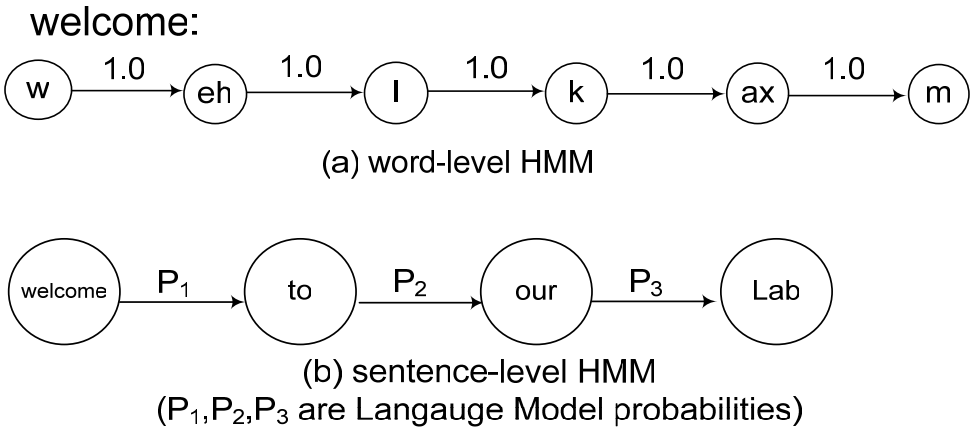

Figure 2.4. HMM can be extended to word-level (a) and sentence-level (b).

Speech is a very complex signal with many variants. **Co-articulation** is one of the most common variants in speech. It means that acoustic realizations (time/spectral feature)

of a phoneme will be changed with the articulations of neighboring phonemes. To describe the co-articulation phenomena, **context-dependent** (CD) phonemic HMM is usually used in continuous speech recognition. The most commonly used CD HMM model is triphone, which considers only one left and one right context of a monophone[4]. Different triphones with the same kernel phone are called **allophones** [Deng&O'Shaughnessy 03]. For example, *aa-b+k* (which means the left context is *aa* and the right context is *k*), *ae-b+k*, *ae-b+t*, …., are called the allophones of monophone *b*. The total number of triphones is a large one considering the commonly used monophonic set including about 40 phonemes. If the position-dependency are also considered which distinguish crossword triphones from within-word triphones (e.g., within-word triphone "*eh-k+ax*" in "re*c*ognition" is different from crossword triphone "*eh-k+ax*" in "wrec*k* a good ship"), this model is called "**position-dependent context-dependent (PD-CD) model**". The total number of PD-CD triphones (logical triphones) will be doubled compared with regular non-PD CD triphones without position dependency. For example, in our experiments a 44-mophone set is used, which resulted in 77,660 non-PD and 151,748 PD logical triphones. Training data are always insufficient for reliable estimations of the parameters of HMM models of so many triphones, especially when Gaussian Mixtures are used as HMM's output model. However, not every triphone appears in speech with same frequency, and some triphones are seldom seen in training or

[4] Similarly, "quin-phone" considers two monophones in both left and right contexts, "septa-phone" considers three monophones in both left and right context, but they are used in much less infrequency than "triphone".

testing data. Some clustering algorithms such as **decision tree based clustering** [Hwang&Huang 92, Hwang+ 93] can be used to reduce the number of physical triphone models to be trained, and a large portion of logical triphones thereafter are shared by or tied to those physical triphones. For example, after decision tree based clustering, the before-mentioned triphones in our experiments are reduced to only 18,836 for non-PD and 38,812 for PD models.

2.1.3 Language Model

Language Model (LM) is to deal with this problem: given a sequence of previously spoken words, what is the probability that the word *w* will be spoken next? There are different ways to attack this problem, including Context-Free-Grammar (CFG) [Tomita 87], Stochastic CFG [Lari&Young 91], and *N*-gram model [Jelinek 91]. CFG-based methods try to use a set of knowledge-based rules to define the production of a sentence in words, and N-gram LM uses a set of counting-based probabilities to predict the next word. Although the first method seems more reasonable and closer to grammar rules, the second method is much more successful in reality because the knowledge-based rules are too complex to be represented by a grammar model such as CFG, whereas *N*-gram LM can be easily obtained and consistently integrated with acoustic model in a stochastic framework based on HMM.

An *N*-gram Language Model can be simply represented by $P(w_n \mid w_{n-N+1},...,w_{n-1})$,

where $w_{n-N+1},...,w_{n-1}$ are *N*-1 words that appeared immediately before current word w_n. By using the fundamental theorem of probability, N-gram probability can be easily estimated using a large text corpus. For those words appeared not frequently enough in the corpus, some **smoothing** techniques are needed to obtain as confident estimations as those common words [Church&Gale 1991, Kneser&Ney 95, Chen&Goodman 98]. **Backing-off** model [Katz 87] is one of the most commonly used smoothing techniques[5]. The idea is to use lower-order N-gram to approximate the probabilities of those uncommon words. In this model the N-gram probability is expressed as follows:

$$P(w_n \mid w_{n-N+1},...,w_{n-1}) = \begin{cases} P(w_n \mid w_{n-N+1},...,w_{n-1}) = \dfrac{Count(w_{n-N+1},...,w_{n-1},w_n)}{Count(w_{n-N+1},...,w_{n-1})}, \\ \quad \text{if } (w_{n-N+1},...,w_{n-1},w_n) \text{ exists or appears frequently enough;} \\ \alpha(w_{n-N+1},...,w_{n-1})P(w_n \mid w_{n-N+2},...,w_{n-1}), \quad \text{otherwise.} \end{cases} \tag{2.3}$$

where $\alpha(w_{n-N+1},...,w_{n-1})$ is referred to as the back-off coefficient of *N*-1-gram $P(w_n \mid w_{n-N+2},...,w_{n-1})$, and its main usage is to make the total probability mass of the *N*-gram equal to 1.

The most commonly used *N*-grams are bigram and trigram, where *N* equals to 2 and 3, respectively. Higher order *N*-grams such as 4-gram are also used in some research systems [Woodland+ 02], with much more training data required. Language Model and its fast

[5] Class-based Language Model is another alternative when word-based N-gram is not well-estimated. See details in [Huang+01].

lookup are very important in LVCSR decoding, which is one of the main factor affecting the decoding speed, and it will be discussed in later sections.

2.1.4 Pronunciation Dictionary

A pronunciation dictionary defines the phoneme constituents for each word in the vocabulary. Figure 2.5 gives some entries of a dictionary used for Wall Street Journal 20K task. Since stochastic pronunciation model is not used in this task, multiple pronunciations will be regarded as having equal a priori probability.

```
…
ACCEPTS eh k s eh p s
ACCESS ae k s eh s
ACCESSIBLE ae k s eh s ax b ax l
ACCESSORIES ae k s eh s ax r iy z
ACCESSORY ae k s eh s ax r iy
ACCIDENT ae k s ih d eh n t
ACCIDENT ae k s ih d ih n t
ACCIDENTAL ae k s ih d eh n t ax l
ACCIDENTALLY ae k s ih d eh n t ax l iy
ACCIDENTALLY ae k s ih d eh n t l iy
ACCIDENTS ae k s ih d eh n t s
ACCIDENTS ae k s ih d ih n t s
….
```

Figure 2.5. Part of pronunciation dictionary for WSJ 20K task.

2.1.5 Decoding Engine

As mentioned in Section 2.1.1, a sequential decoding procedure is used to obtain an optimized word sequence with the highest MAP probability, as defined by formula (2.1). The decoding component of speech recognition is also called decoding engine, which is the kernel of a LVCSR system, because it combines all knowledge sources including acoustic model, language model, pronunciation dictionary, decoding algorithm and determines the overall performances of LVCSR.

There are different ways to classify decoding engines based on one or more features of decoding algorithms [Aubert 02, Ney&Ortmanns 99]. Some of them are listed as follows:

Breadth-first vs. Depth-first: breadth-first search expands search paths with equal depth and depth-first search processes along one path until reaching a depth bound and then considers alternative paths. For example, Viterbi algorithm is breadth-first and A* algorithm is depth-first.

Time-synchronous vs. Time-asynchronous: the search hypotheses are formed in a time-synchronous or time-asynchronous fashion over the sequence of acoustic vectors.

One-Pass vs. multi-pass: the word sequence ouput is produced from one or more passes of search over the input sentence.

Word-conditioned vs. time-conditioned: each search hypothesis is conditioned on the predecessor word or the ending time of the predecessor word. Word-conditioned search

is more popular than time-conditioned search.

Single-best vs. N-best or word lattice (or word graph): the output is only the most likely word sequence, or top N possible word sequences, or a word lattice organized in a directed graph.

Dynamic search network vs. static search network: the search space is dynamically generated during search or pre-compiled with all knowledge sources before search, such as Weighted Finite State Transducer (WFST) [Mohri+ 02].

Linear vs. tree lexicon: the search space is based on linear lexicon or pronunciation-prefix tree organized lexicon.

Tree-copy vs. single-tree: in the dynamic search network and lexical tree based decoding algorithm, the hypotheses of different language contexts are organized in multiple tree copies or in single lexical tree; in other word, the language context information is distinguished at tree-level or node/arc-level.

Cross-word vs. within-word: the search space does or does not consider cross-word triphone, i.e., at each word end, whether or not to create those triphones considering right context as the first phoneme of the next possible word.

Although different combinations of above features will generate many possible decoding engines, in fact only a few of them really work for LVCSR. In the later part of this section, three algorithms and some of their combinations are described: Viterbi time-synchronous beam algorithm, A* time-synchronous algorithm and multi-pass

based algorithm.

2.1.5.1 Viterbi Time-Synchronous Beam Search

Viterbi beam search is based on the **Dynamic Programming** (DP) algorithm [Bellman 57]. It is firstly introduced into speech recognition by CMU's HARPY system [Lowerre 76] and BBN's BYBLOS system [Schwartz+ 85]. It was first successfully used in LVCSR by Hermann Ney's research group in Philipis Research Lab [Ney&Ortmanns 92] and RWTH University of Technology [Ney&Ortmanns 99], both in Aachen, Germany. Dynamic Programming is a sequential optimization algorithm to decompose a problem into some sequential, independent sub-problems, and by recursively solving those sub-problems, it will get the final solution of the original problem in a bottom-up way ([Cormen+ 01]). For example, in speech recognition, the optimal word sequence in a sentence from time 1 to time *t* is obtained by recursively getting the optimal word sequence of HMMs from time 1 to *t*-1 until the *t* reaches the zero. In word-conditioned search algorithm which is commonly used for LVCSR, Dynamic Programming is modified to recursively find the best previous word for the current time *t*. The decoding procedure can be decomposed into two steps:

Forward-extension: All possible hypotheses (paths) are extended from time 0 to time *T*-1 where *T* is the number of acoustic vectors (frames) in a sentence. During the extension, path scores are accumulated by combining the acoustic score and language score for all acoustic vectors (speech feature vectors) up to current frame and at each time each path

will record its best previous word when a new word is created. Different heuristic pruning and lookahead methods may be used at this step to cut out those unpromising search paths to speed up the decoding.

Backtrace: After the last frame at *T*-1 is processed, a best path with the highest score is selected, and then a backtrace is conducted by recursively getting the best previous word of current word recorded in the forward-extension step.

Besides above steps provided by DP algorithm, an important approximation is needed, that is, the best word sequence can be approximated by the best state sequence of phoneme HMMs underlying the current sentence. This approximation is called **Viterbi Approximation**, and it can be expressed as follows [Ney&Ortmanns 99, Sixtus 03]:

$$
\begin{aligned}
[w_1^N]_{opt} &= \arg\max_{w_1^N, N} P(w_1^N \mid o_1^T) = \arg\max_{w_1^N, N} \left\{ P(w_1^N) \cdot P(o_1^T \mid w_1^N) \right\} \\
&= \arg\max_{w_1^N, N} \left\{ \prod_{n=1}^{N} P(w_n \mid w_{n-m+1}^{n-1}) \cdot \sum_{s_1^T} \prod_{t=1}^{T} \left\{ P(o_t \mid s_t, w_1^N) \cdot P(s_t \mid s_{t-1}, w_1^N) \right\} \right\} \\
&\approx \arg\max_{w_1^N, N} \left\{ \prod_{n=1}^{N} P(w_n \mid w_{n-m+1}^{n-1}) \cdot \max_{s_1^T} \prod_{t=1}^{T} \left\{ P(o_t \mid s_t, w_1^N) \cdot P(s_t \mid s_{t-1}, w_1^N) \right\} \right\}
\end{aligned}
\tag{2.4}
$$

where s_1^T is the state sequence of phoneme HMMs corresponding to the acoustic vector o_1^T and word sequence w_1^N (*N* is the word number in the sentence). In the second line of (2.4), the best acoustic score has to be computed by summing over all those possible paths with different state sequences, which involves exponentially increased search space with t and in most cases it is impossible to be handled for large vocabulary task. However, by

Viterbi approximation, the summation over all possible state sequences is replaced by a maximization (the third line of (2.4)), which reduces the search space by a large factor. Actually, recognition experiments have proved that this approximation brings almost no loss to word accuracy [Demuynck+ 02].

Viterbi algorithm utilizes **time-synchronous** search, which means that at each time interval, the path scores for different hypotheses are computed based on the same acoustic data, and so heuristic pruning can be easily operated at each time interval, e.g., by comparing those different path scores and only keeping the most promising ones. **Beam pruning** is one of the most commonly used heuristic pruning approaches, where a beam value is set as the maximal difference between the highest path score and other competing path scores, so that all other paths falling beyond this beam will be pruned out. For LVCSR, the experimental results show that only a small percentage of the entire search space (the beam) needs to be kept for each time interval without increasing error rates. According to different level of knowledge sources such as HMM states, HMM phones, and words, it is possible to set different beams for fast decoding. This will be discussed in detail in Section 2.3.1.

A detailed mathematical formula of Viterbi time-synchronous beam search based on trigram language model and within-word HMM is given as follows [Ney&Ortmanns 99; Huang+ 01]:

Define: 1) $Q_{uv}(t,s)$ = Probability of the best partial search path that ends at time t in

state *s* of language model history (*u*, *v*).

2) $B_{uv}(t,s)$ = Backtrace pointer for the best partial search path ending at time *t* in state *s* of language model history (*u*, *v*).

3) $A(w,t,Sc,B)$ = Backtracing array with four members at each entry: current new word *w*, current time *t*, path score *Sc* and the backtrace pointer *B* to the previous word *v*.

Initialization: For all the word *u* which can start a sentence：

$$Q_u(0,0) = 0$$

$$B_u(0,0) = -1$$

Induction: For time *t* = 0 to *T*-1, do

1) For all active words do

Intra-word transitions according to: (**state-level recombination**)

$$Q_{uv}(t,s) = \max_{\sigma}\{P(o_t, s \mid \sigma) \cdot Q_{uv}(t-1,\sigma)\}; \tag{2.5}$$

$$B_{uv}(t,s) = B_{uv}(t-1, \sigma_{uv}^{\max}(t,s)), \tag{2.6}$$

where $P(o_t, s \mid \sigma)$ is the product of the probability for the HMM transition from state σ to state *s* and the emission probability of state *s* at time *t*, i.e.,

$$P(o_t, s \mid \sigma) = P(\sigma \mid s, w) \cdot P(o_t \mid s, w) \tag{2.7}$$

where *w* is the current possible word corresponding to state *s*.

Also $\sigma_{uv}^{\max}(t,s)$ is the best predecessor state of state *s* in current LM context (*u*, *v*) at time *t*.

2) For all active word-final states do

2.1) Inter-word transitions according to: (**word-level recombination)**

$$Q_{vw}(t,s_0) = \max_u \{P(w \mid u,v) \cdot Q_{uv}(t,s_w)\}; \tag{2.8}$$

2.2) Store the best predecessor word u_{max} and boundary t into backtracing array by generating a new item in this array: $(w,t,Q_{vw}(t,s_0),B_{u_{\max}v}(t,s_w))$.

2.3) Store the backtracing pointer into this partial path as:

$$B_{vw}(t,s_0) = ind(w,t,Q_{vw}(t,s_0),B_{u_{\max}v}(t,s_w)) \tag{2.9}$$

In above three steps, s_0 denotes the first state for possible new word, in lexical-tree based search space as to be introduced in Section 2.1.6, it is the root arc of lexical tree for the next new model. It is a fake state which does not emit a feature vector [Ortmanns+ 97a]. s_w is the final HMM state of word w. $ind(w,t,Q_{vw}(t,s_0),B_{u_{\max}v}(t,s_w))$ is the index of backtracing array according to word w, predecessor words $u_{\max},v$ and boundary t. Note that the score for this new backtrace item is $Q_{vw}(t,s_0)$, and it has a backtracing pointer to its processor word: $B_{u_{\max}v}(t,s_w)$.

3) Pruning: Find the score for the best path and decide beam threshold, prune unpromising hypotheses.

Termination: Pick up the best path from all the possible final state s of search space at time T-1, obtain the optimal word sequence according to $B_{uv}(t,s)$ and backtrace array.

The crossword HMM based Viterbi search algorithm is very similar to above

algorithm, only two formulae (2.8) and (2.9) need changes. But the actual search space and computation will be much larger than within-word based Viterbi search, which will be described in Chapter 3.

2.1.5.2 A* Time-Asynchronous Search

A* algorithm is a famous heuristic search algorithm in Artificial Intelligence [Nilsson 98]. Its application in speech recognition was first introduced by IBM [Bahl+ 83] and was successfully used in IBM's LVCSR systems [Bahl+89, Gopalakrishnan+ 95]. The main principle of A* search is using a heuristic function to guide the sequential search, and the heuristic function is composed of two parts as below:

$$f(n) = g(n) + h(n), \tag{2.10}$$

where n is any node in the search graph or search space, and in speech recognition, n stands for any word in the search network, $g(n)$ is defined as the heuristic score from the beginning node to current node and $h(n)$ is defined as the heuristic score for the rest part of possible path, i.e., from current node to the ending node. The main steps of A* algorithm is described as below [Huang+ 01]:

1) Put the starting node n_0 in a list called OPEN;

2) Generate an empty list called CLOSED;

3) If the OPEN list is empty then the algorithm fails and exits.

4) Pop out the first node n in OPEN, move it into CLOSED list.

5) If n is the target node, then the algorithm succeeds. Output the path back-traced from n to n_0 as the best path and exits.

6) Extend node n, all its successor nodes are put into a set called as Successor Set of n, or $SS(n)$, Be sure to eliminate the ancestors of n from $SS(n)$.

7) For any node $v \in SS(n)$ do

7a) If $v \in$ OPEN and the accumulated score of the new path is larger than that for the one in the OPEN list, do

i) Change the traceback (parent) pointer of v to n and adjust the accumulated path score for v.

ii) Evaluate $f(v)$ for v and goto Step 8.

7b) If $v \in$ CLOSED and the accumulated score of the new path is larger than the partial path ending at v in the CLOSED list, do

i) Change the traceback (parent) pointer of v to n and adjust the accumulated score and heuristic function f for all the path containing v.

ii) Goto Step 8.

7c) Create a pointer pointing to n and push it into the OPEN list.

8) Reorder the OPEN list in descending order of the heuristic function $f(n)$.

9) Go to Step 3.

It can be proved [Nilsson 98] that if the heuristic function $h(n)$ of estimating the

remaining score from n to goal node G is not an underestimate of the true score from n to G, then the A* algorithm is guaranteed to find the best path (or best word sequence in speech recognition). The informal proof can be shown as follows [Huang+ 01]. When the top most node in the OPEN list is the goal node G in Step 5, it can be obtained that

For any $v \in$ CLOSED, $f(v) \leq f(G) = g(G)+h(G)=g(G)$. (2.11)

which means the score of any incomplete path are not larger than the first found complete path. Since the score for any incomplete path is overestimated, the first found complete path in Step 5 must be the optimal path. A similar point can be used to prove that the Step 7b is actually not necessary for A* search, i.e., there can not be another path with a larger score from the starting node to a node that has been expanded.

A search algorithm is called admissible if it can guarantee to find the best path. A* search algorithm is an admissible algorithm when $h(n)$ function is an overestimated score function. In contrast, the Viterbi beam search is not an admissible search since it can not guarantee in theory to find the best path.

A* search is also a time-asynchronous search because the scores of different paths in OPEN list are computed based on different lengths of acoustic data, which is different from time-synchronous Viterbi beam search.

When the heuristic function is close to the true remaining path score, the search can usually find the optimal solution without too much effort. If a heuristic function h_1 is smaller than another heuristic function h_2 in every nodes of the search network, (and h_1 is

still the admissible heuristic function), it is said that h_1 is more informed than h_2. One of the main problem of A* algorithm is how to find an informed heuristic function, which is usually a very difficult problem.

A* algorithm is also a best-first algorithm which means that the best path is always the first output without considering all other incomplete paths. In contrast, the Viterbi beam algorithm is a breadth-first algorithm in which the best path will be obtained at the same time as all other competing paths[6]. Based on this difference, it seems that when A* algorithm is used in continuous speech recognition, it may save computation significantly by not keeping all other unpromising paths. Actually, A* search is not as efficient as Viterbi beam search in LVCSR. The main reasons exist in two folds:

Firstly, informed heuristic function $h(n)$ is not easy to obtain. Usually used heuristic functions are much larger than the true remaining score for some portions of the search graph. When those less informed heuristic functions are used, the search effort will be largely increased. An extreme example is $h(n) = 0$, which is a very little informed admissible function. In this case, the A* search actually tends to be an exhaustive search since longer path always means lower score with the accumulation of log probabilities in forward score of $g(n)$. In this way, the search will be very slow since almost all word hypotheses need extensions at each time.

Secondly, it is not easy to use beam pruning to reduce the unpromising paths in A*

[6] However, the implementation of N-best Viterbi Time-Synchronous search is not trivial, because the extension of N-best list will result in exponential growth of search space. Please see [Huang+ 01] for the details.

search because of its time-asynchrony. Since the scores for all hypotheses are accumulated based on different lengths of acoustic data, it is difficult to do beam pruning in the same way as in Viterbi beam search. If some other heuristic methods are used to obtain comparable scores for different hypotheses [Huang+ 01], the optimality of the search may be sacrificed by pruning out the best path.

Although A* algorithm is not as efficient as Viterbi beam search for one-pass search of LVCSR, it is important for multiple-pass search and word-lattice (or word-graph) based search. This will be presented in next Section.

2.1.5.3 Multipass based Search

Multipass based search means that the search algorithm will compute the search scores based on multiple accesses of the acoustic data. It is often used in the paradigm where a simple knowledge source is used in the first pass and a N-best list or word-lattice (graph) is firstly generated, and then in latter passes, more precise and complicated knowledge source can be used to reorder N-best hypotheses obtained in the first pass. The shortcoming of multipass search is the speed problem, since it will always wait until the end of a sentence before the second or latter pass can be operated, and multiple computation of path scores will result in longer time than one-pass computing. Although the second pass or latter pass may be conducted in a very fast speed since the search space is reduced to the N-best list or word-lattice(graph) given by the first pass, there are common concerns about the

non-admissible pruning effect in the first pass due to the use of insufficient knowledge source. However, there are also some justifiable reasons for the usage of multipass based search.

Firstly, precise knowledge sources may be too expensive to be used in the one-pass search. For example, higher-order N-gram (such as 4-gram or 5-gram) and long-distance context-dependent models (such as quin-phone or septa-phone, which considers more than one left or right contexts for each allophone) are very expensive in model size and in management of search space if they are used in one-pass search. By using multipass search, those expensive language models or acoustic models can be easily applied to N-best list or word-lattices (word-graphs).

Secondly, N-best lists or word-lattices (word-graphs) generated in multipass strategy is a very good interface for some applications such as speech understanding or integration of recognition results from multiple models. For example, in DARPA-funded benchmark testing of Broadcast News and Switchboard tasks [Pallett 02], a method called ROVER [Fiscus 97] is commonly used to combine recognition results obtained by different models, which is based on a fusion and rescoring of word-lattices generated from former passes using different models. It is reported that by ROVER combination, the word error rate can be reduced by about 6%~7% [Fiscus 97].

As mentioned before, A* search is very useful in multipass based search strategy. One reason is that A* search can be used in the second-pass based on the word-lattice

generated by the first pass. At this time, A* search is conducted from the word-ending to the word-start, and a very good (informed) heuristic function $h(n)$ can be easily obtained by using the path scores from the start node to current node. This score has already been computed in the first pass using simpler acoustic model and language model on real observation data. In this way, the best word sequence is much more likely to be obtained in backward A* search than one-pass A* search. This search algorithm is also called **two-pass forward-backward** search. The second reason is that if A* search is used in the first-pass, then it is very easy to output the N-best list by selecting $h(n)$ to be equal to negative infinity ($-\infty$) for ending node. In this way, the search will not end as usual A* search until all possible word sequences are produced.

2.1.6 Lexical Tree Based Search Space

In the state-of-the-art LVCSR decoding engines, tree-based structure is usually used to represent the lexical knowledge and search space. This structure is called **Pronunciation Prefix Tree** (PPT) [Alleva+ 96] or **Lexical Prefix Tree** (LPT) [Ney&Ortmanns 99] because equal word prefixes are merged in the tree. Especially for large vocabularies containing several thousands of words there are many words sharing the same prefix. The highest ambiguity during the processing of competing word hypotheses in search is encountered within the first phoneme of the words. Thus the reduction of redundancy especially at the beginning of the words by prefix merging results in a considerable

reduction of the number of active search hypotheses [Ney&Ortmanns 99, Sixtus 03].

An example of PPT is shown in Figure 2.6. Each arc of the tree represents a triphone and the corresponding HMM respectively (One can also use nodes to represent triphones). A path from the root of the tree to a leaf composes the HMM of a complete word of the vocabulary. Due to the prefix merging the identity of the corresponding word is not known until at the leaves of the tree. Therefore, when hypothesizing a partial word sequence during search, the language model probability of the last word of the sequence can not be integrated into the probability of the partial word sequence hypothesis until the leaf of the lexical prefix tree is reached. This language model probability depends on the language model history of the search hypothesis (i.e., the (n-1) immediate predecessor words). In order to retain the information about the history of a search hypothesis until a tree leaf is reached, different competing search hypotheses have to be kept separately according to their language model history.

In several reported systems, search hypotheses from the same language model history are linked together into a so-called **tree copy**. Each tree copy is marked with the corresponding language model history (or called **linguistic history**). All search hypotheses traversing the same tree copy are guaranteed to have the same linguistic history. In Figure 2.7, an example of tree-copy based search network is illustrated for a three word vocabulary (*A, B, C*) where a bigram language model is used. For each possible linguistic history a separate copy of the PPT is provided, which is depicted in a simplified

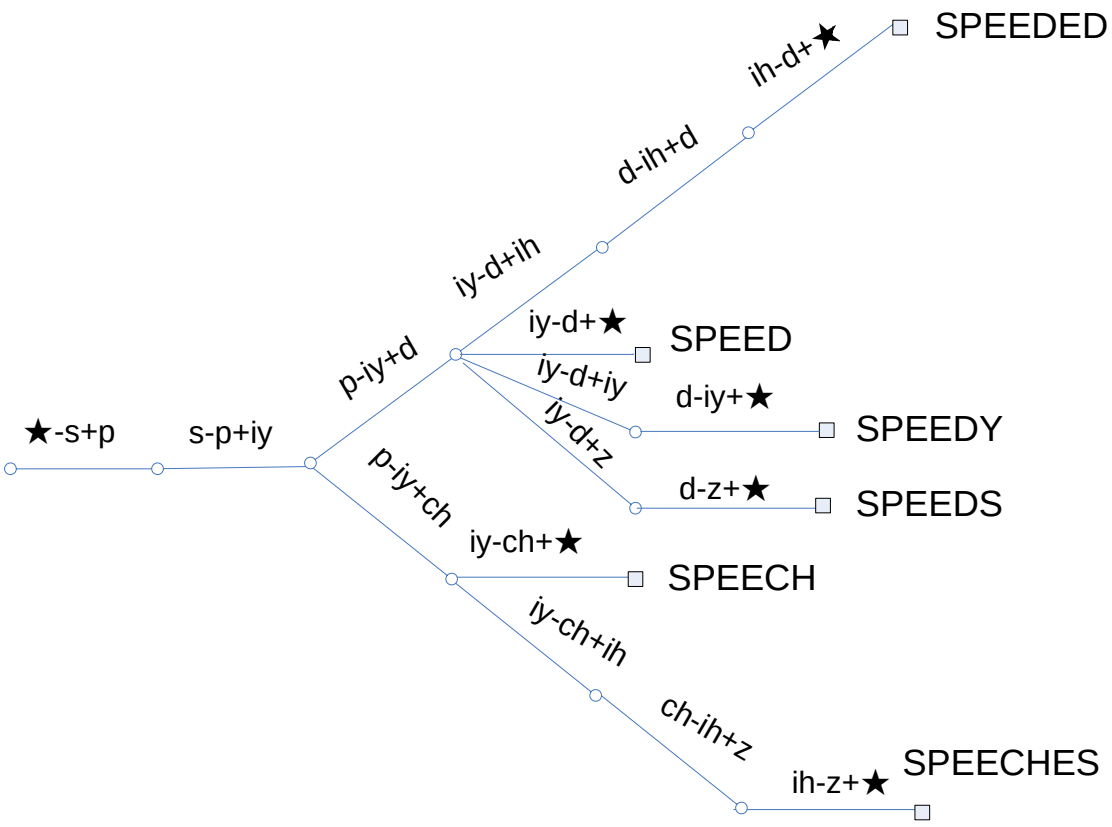

Figure 2.6. An example of a lexical prefix tree. The arcs represent the triphones of the words and the corresponding HMMs respectively. ★ is a special context representing a word boundary. Each path from the root of the tree to a leaf composes one word of the vocabulary.

skeleton form. In addition, a separate copy is needed for the silence word ($) which is necessary for the beginning and ending of a sentence. When a search hypothesis reaches a leaf arc of a tree copy, the last word of the partial word sequence hypothesis is known and the hypothesis probability can be enhanced by the language model probability. The linguistic history of the hypothesis is updated accordingly and the hypothesis is traversed

to the root arc of the tree copy for the corresponding linguistic history. For example, when a hypothesis path from a linguistic history *A* reaches the end of word *B*, it will traverse to a new tree copy with linguistic history of *B*. This is illustrated by the dashed lines in Figure 2.7. At the root arcs of the tree copies the **recombination at word level** (see formula 2.8) is conducted, i.e., only the most probable hypothesis reaching a tree copy's root arc is considered for further extension, and the others are discarded (because exactly the same acoustic model score and language model score will be added for all those hypothesis paths in the future search). Using this *n*-gram word conditioned **re-entrant** network [Aubert 00] all possible word sequences can be represented. The potential size of the search network grows exponentially with the length of the linguistic history. However, due to **beam pruning** and **word-level recombination** only a small portion of this huge network will be considered during search. As a result, in tree-copy based decoding engine, the search network is not compiled completely and statically beforehand but is constructed dynamically during the runtime of the search. At each time interval only that part of the network which contains active search hypotheses is kept in memory. If a search hypothesis is pruned, it will be removed from memory. If a search hypothesis reaches a part of the network which is not already present in the network, this part is created and integrated into the network on demand. ([Sixtus 03])

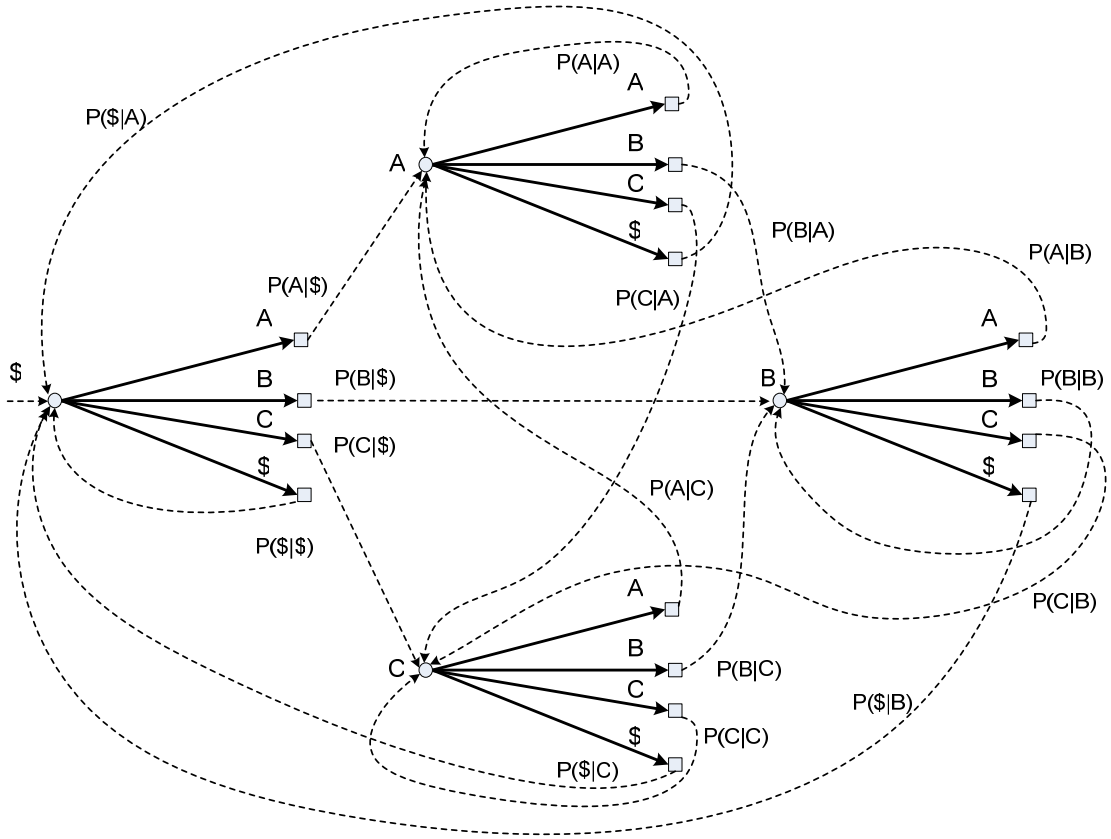

Figure 2.7. Search network based on lexical prefix trees and a bigram language model. The thick lines represent within-word transitions, the dashed lines represent crossword transitions.

Above descriptions of search organization is based on so-called **word conditioned tree search** [Ortmanns+ 97a, Ney&Ortmanns 99]. An alternative organization of the search network using lexical prefix trees is the use of **time conditioned** tree copies [Ortmanns+ 96b]. In this case search hypotheses with the same start time of their last word hypothesis are linked together. This potentially produces a smaller number of active tree copies but the recombination at word level is much more complex than in the word

conditioned search. In [Ortmanns+ 96b] it is shown that the two search algorithms are equivalent and that the word conditioned search can be derived from the time conditioned search analytically by optimizing over boundary times of words.

2.2 Performance Measurements of LVCSR

2.2.1 Word Accuracy (Word Error Rate)

In speech recognition, the word error rate (WER) is defined as follows:

$$WER(\%) = \frac{Substitutions + Deletions + Insertions}{\text{Number of Words in Reference}} * 100\% \qquad (1.12)$$

where *Substitutions* is number of the cases where word A is recognized as another word B, *Deletions* is the number of cases where word A is missed in the output, and *Insertions* is the number of cases when word B in the output is a spurious word which does not correspond to any word in the reference.

Word Accuracy (WA) is defined as

$$WA(\%) = 1 - WER(\%) = \frac{Matches - Insertions}{\text{Number of Words in Reference}} * 100\% \qquad (1.13)$$

where *Matches* is the number of words in the output which match the words in the reference.

In the calculations of word accuracy, a dynamic programming based procedure [Huang+01] is carried out to align recognition result and reference, and the values of *Substitutions*, *Deletions*, and *Insertions* are the output of the alignment.

2.2.2 Speed (Real-Time Factor)

The speed of a decoding engine is defined as the time for decoding of an utterance measured by the time for speaking of the sentence, or called Real-Time Factor (RTF), i.e.,

$$\text{Speed (xRT)} = \frac{\text{Time of Decoding one Sentence}}{\text{Time of Speaking the Sentence}} \tag{2.14}$$

It is worth noting that in the work of this dissertation the decoding time is measured by **real elapsed time** instead of **CPU-time** which was used in a previous work in Computer Science department of CMU [Woszczyna 98]. CPU-time is the time when the recognition process has exclusive use of the central processing unit of a computer with a multitask operating system such as UNIX or Windows Server 2003. The difference between CPU-time and real recognition time is that network access or loading virtual memory pages from disk is not counted in CPU-time. The advantage of using CPU-time is that they are independent of the amount of core memory available and the work load that other processes put on the CPU, making comparisons between separate test runs easier. However, using real recognition time is more meaningful since it is the real speed performance of the decoding system. For fair comparisons of speed between different test conditions, each time only one user process (decoding) is running in the whole system.

In the following chapters, the recognition speed is always given by Real-Time Factor as in (2.14).

2.2.3 Memory Cost

Memory cost (in Mega Bytes or MB) is also an important measurement for the system performance of a decoding engine. In our experiments, there are two different ways to measure the memory cost of a decoding task. The total memory cost for a decoding process is obtained by recording the “Peak Memory Usage” from the Task Manager of Windows Server 2003. As mentioned above, each time only one decoding task is running and the physical memory of the system is large enough to avoid unnecessary virtual memory swapping. But for the memory cost of different components of a decoder such as Language Model lookup and its sub-components, it is difficult to get the true allocated memory space for them by programming, therefore theoretical values are used by accumulating the amount of bytes of each data structure. Obviously, theoretic values are not equal to the true memory space allocated by the operation system (mainly based on page management mechanism) but it is roughly proportional to the latter.

It is preferable to make the memory cost of a decoding engine as small as possible. Too much memory cost means the system is difficult to be deployed into commonly affordable systems or memory-limited system such as PDA, Smart Phone, Pocket PC, etc. However, in most of previous research of LVCSR, memory cost is seldom cared. Many complex algorithms obtained performance improvement at the expense of higher memory cost, such as higher-order language models and long-term context dependent acoustic models. With

the order of language model increases, the total number of *N*-grams increases from tens of thousands to tens of millions, which dramatically increases the memory cost for the decoding system. Same tendency is also seen in context-dependent acoustic models.

Accuracy, speed, and memory cost are correlated and interact with each other. For example, the improvement of accuracy is commonly obtained at the expense of higher memory cost and slower speed. This is true for a long time by the use of more complicated models of higher-order language models (such as trigram, 4-grams) and long-term context dependent acoustic model (such as triphone, quin-phone, septa-phone, etc.). On the other hand, many proposed methods to speed up decoding were at the expense of higher memory cost and most of time degraded accuracy. It is a common experience that certain trade-offs are necessary among the three performance measurements. However, in this work, one of main contributions of the author is to propose a new LM lookup algorithm which has good properties in all three aspects, i.e., largely speeding up the decoding, significantly saving memory and with no degradation to the accuracy.

2.3 Fast Decoding of LVCSR

The straightforward implementation of Viterbi algorithm as described in Section 2.2 will not deliver real-time performance for LVCSR tasks. There are many proposed methods to speed up this algorithm. Some commonly used techniques for fast decoding of LVCSR is

introduced in this section.

2.3.1 Heuristic Pruning

As described in Section 2.1.6, the potential size of the search network in lexical tree based search organization grows exponentially with the length of language model history**,** but by using heuristic pruning such as beam pruning, the search space can be reduced to a very small size with little or no loss of word accuracy. In Viterbi search algorithm, heuristic pruning is very easy to carry out because of its time-synchrony. Several different heuristic pruning methods are described as follows:

State-Level Beam Pruning: This pruning can be carried out on the state-level hypotheses of phoneme HMMs in each time interval. It first finds the highest path score $\max_s Q_s(t)$ for current state-level hypotheses, and then prunes away those hypotheses if their scores $Q_s(t) \le \max_s Q_s(t) - b_s$, where b_s is the state-level beam. This pruning was also called as "**Acoustic Model Pruning**" [Ortmanns+ 97b] because it usually occurs at the internal nodes of lexical tree based search and the language model knowledge has not been integrated into the path score at the pruning time. As will be presented in later part of this section, actually in many systems, approximate language model score (LM lookahead score) can also be integrated into path score for internal nodes. Based on this point of view, "State-level Beam Pruning" may be more suitable.

Word-Level Beam Pruning: This pruning is carried out at the word-ending nodes

after new word hypotheses are created and before the new backtrace items are generated (see Section 2.1.5.1). It first finds the highest path score $\max_w Q_w(t)$ for current word-level hypotheses, and then prunes away all those word candidates if their scores $Q_w(t) \le \max_w Q_w(t) - b_w$, where b_w is the word-level beam. This pruning is also called "**Language Model Pruning**" [Ortmanns+ 97b].

State-Level Histogram Pruning: It is also carried out at state-level and used to limit the maximal number of active state hypotheses. It first re-orders all those state hypotheses by their path scores and then keeps only the highest N_s hypotheses and prunes out all lower-scored hypotheses ([Steinbiss & Tran+ 94]). N_s is called state-level histogram threshold.

Word-Level Histogram Pruning: Similar to state-level histogram pruning, this pruning is carried out at word-level and limits the maximal number of active word candidates generated at each time frame. N_w is used to represent word-level histogram threshold or called word cut-off threshold.

Crossword Language Model Lookahead: this is a pruning method specifically used for crossword triphone based lexical tree search, which will be described in Section 3.3. The corresponding beam is represented by b_{xwd}.

Phone-Level Beam Pruning: This pruning is very similar to state-level beam pruning and word-level beam pruning, and it is carried out at phone (HMM) level. The corresponding beam is represented by b_p. As to be mentioned in Chapter 3, in our

system, this beam pruning is not used since no advantage will be gained by phone-level pruning on top of state-level and word-level beam prunings [Huang+ 01].

There is no effective mathematical method to estimate the values of all those beams and histogram thresholds before a decoding engine is operated formally. Empirical tuning by repeating the decoding experiments using some small set of data is the only method to find the optimal values. These data used for system tuning is called "**development testing data**", which is different from **training data** and **evaluation testing data**. However, there are some common qualitative relationships between beams at different levels. For example, since the word-level pruning will be more reliable than phone-level pruning and state-level pruning because more knowledge is integrated, the word-level beam is usually smaller than phone-level beam, and in return, the phone-level beam is usually smaller than state-level pruning beam. Same relationship also applies to histogram thresholds. As a summary, the followings relationships are usually correct,

$$b_w < b_p < b_s \tag{2.15}$$

$$N_w < N_s \tag{2.16}$$

Experience shows that the maximal number of word candidates (N_w) at each time frame may be set as small as 15 or 30 and it is still enough for a vocabulary size as large as 33,000. On the other hand, however, the maximal number of active state-level hypotheses have to be kept in the order of 10,000 or 100,000 for the same level of vocabulary

size. In this sense, it may be more suitable to change (2.16) as

$$N_w << N_s \tag{2.17}$$

2.3.2 Language Model Lookahead

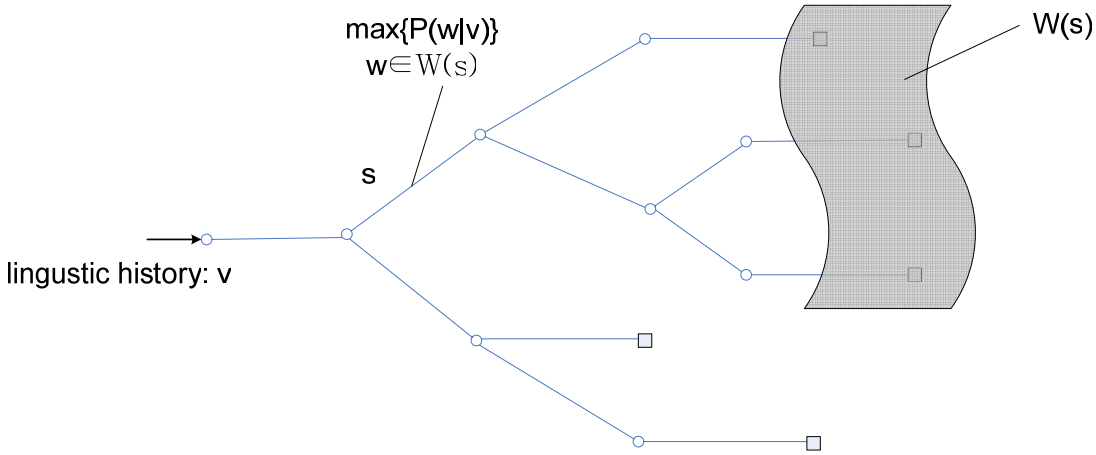

Figure 2.8. Predict the LM scores by maximizing all the LM scores of those reachable words of current node.

As mentioned in Section 2.1.6, in the lexical tree based search organization, the word identities can not be known until the search reaches the word endings. Therefore language model score can not be directly integrated into the scores for those paths in the internal arcs of lexical tree. This may cause search errors because state-level pruning (beam pruning and histogram pruning) in these arcs will be based on insufficient knowledge without considering current language model scores, especially when aggressive pruning is used for fast search speed. Language model lookahead (LMLA) [Ortmanns+ 97b] is a

technique to overcome this problem. The idea is to use the language model probability of the most probably reachable word of an arc as an upper estimate of true language model probability and integrate it into the score of the current search hypothesis whenever a new arc in the tree copy is activated. Figure 2.8 illustrates the procedure of obtaining this probability for a lexical node. This probability is called LMLA score[7]. This early consideration of language model score allows more efficient pruning during beam search and therefore reduction of the number of active search hypotheses.

Since LMLA occurs at each lexical arc for each tree copy, the computation of LMLA is very large especially when high order language model such as trigram and four-gram is used. For example, for a LVCSR task with 33K vocabulary (such as Switchboard task reported in Chapter 4), the number of nodes in lexical tree is 201K. When the average number of new nodes generated (thus the number of LMLA lookups) in each frame is 1000 (which is a common number for such large vocabulary size), for a sentence of 2 seconds duration with a frame shift of 10ms, the total number of LMLA lookups will be 2*1000*1000/10 = 200,000. Considering that each LMLA lookup may involve a series of N-gram lookups (each lookup may be as slow as several microseconds for high-order N-gram and the number of lookups may be as large as several thousands especially for those nodes in early layers of tree) and a maximization operation, the total time spent in LMLA for this short sentence will be very long. Many speedup methods have been

[7] This is an admissible estimation (over-estimate) if regarding LMLA scores as a heuristic function, as described in Section 2.1.5.2.

proposed to reduce this computation. These methods will be analyzed in Chapter 4 since fast language model lookahead is one of the main focuses of this dissertation.

2.3.3 Phone Lookahead

The idea of phone lookahead (PLA) is to anticipate the probability of future acoustic vectors for a phoneme arc in the pronunciation prefix tree before starting for detailed search. The acoustic probabilities are estimated with simplified context independent (CI) phoneme models and an additional pruning step is performed before the arcs are started in detailed search [Ortmanns+ 97]. This pruning is also called **phone lookahead pruning**. A beam b_{PLA} is used to prune away those new arcs with lower lookahead scores compared with the best lookahead score. PLA is a very effective method to reduce the search space by limiting the number of newly generated lexical arcs and it is implemented in our decoding engine.

2.3.4 Acoustic Likelihood Approximation

As reported by many systems, acoustic likelihood computation by formula (2.2) is the lion's share of total decoding time. To reduce this part of computation, different algorithms are proposed to approximate the Gaussian output probability. VQ-based Gaussian Selection [Bocchiere 93] and Generalized Bucket-Box-Intersection (GBBI) [Woszczyna 98] are two typical examples. The main idea of those two algorithms is to pre-classify all

those Gaussian components of the acoustic model according to their mean vectors and variance matrixes (usually diagonal variance matrix is used to reduce the computation), and each classes is attached by a short list of Gaussian components. During decoding, for each acoustic vector, estimation is firstly made about which class it should be assigned to by using the distance between the acoustic vector and the centric mean vector and variance vector of each class, and the likelihood score will be approximated by using only those Gaussians in the short list of assigned class. For example, the normal number of Gaussian components in a mixture density for a triphone HMM is 16, and the number of Gaussian components in a short-list of VQ-based approximation will be set as 10 or less, controlled by a distortion threshold. The disadvantage of those acoustic score approximation methods is that word accuracy will be sacrificed as the trade-off with speed gain. In this dissertation work, neither of those approximation methods is used, which is based on two reasons:

Firstly, in our system, the lion's share of decoding time is not the acoustic computation part, but the part of language model lookup, because trigram language model lookahead is used to obtain lower word error rate. Even after reducing the time taken by language model lookup from 60~80% to only 10% of total decoding time, the lion's share is still not in this part but in search management.

Secondly, the SIMD-based software-level optimization by using Intel's IPP software (described in next section) already provides the speed-up for acoustic likelihood computation without loss of accuracy.

2.3.5 Hardware and SIMD-based Instruction Optimization

There are several ways in implementing hardware based speed optimization.

Firstly, multi-processor architecture with two to eight multi-purpose processors is becoming more and more common (e.g., Intel's Xeon based symmetrical architecture). But for decoding of LVCSR, it is difficult to parallelize processing one utterance in multiple processors because lookahead techniques are commonly used and because of the sequential nature of decoding algorithms. If the computer has enough memory to decode two utterances at a time, it is more efficient to parallelize decoding over several utterances.

Secondly, some researchers and developers also resort to special purpose hardware such as DSP chips for acoustic likelihood computation. However those architectures are too restricted for research purposes and they are also costly for most commercial applications.

Thirdly, a cheaper method is to utilize optimizations provided by instructions set of general-purpose CPU. SSE2 (Streaming SIMD Extensions 2) [Intel 05] in Intel Pentium III, Xeon or later version of CPUs is such an example. SSE2 is based on Single-Instruction Multiple-Data (SIMD) optimization of instruction set, and it is specially designed for multimedia computing including speech recognition applications. By using SSE2 instruction set, the computation of acoustic likelihood can be speeded up by parallelized computing of four Gaussian mixtures in one instruction, or by parallelized computing of

four dimensional feature (such as MFCC) components in one instruction. Both assembly-level [Li 03] and C-level integration of SSE2 optimization is possible. In our decoding system, a C-level language API package called Integrated Primitive Performance (IPP)[8] is used to reduce time in acoustic likelihood computation.

It is obvious that as the speed of general-purpose CPU increases, the decoding engine running in this CPU will also be speeded up. With the help of Moore's Law, that is, the clock frequency of computer's CPU doubles every 18 months, high speed computers has been an important factor to enable real-time LVCSR. Table 2.1 lists the results of comparative experiments for a LVCSR decoding task (WSJ 20K) obtained on three different Dell Xeon PC workstations with different CPU clock rates and physical memory size. Identical pruning thresholds were used for each case. The operating system is Windows Server 2003 Standard, the compiler is MS Visual C++. The word error rate is fixed as 9.73%. Here the optimization methods included phone lookahead, SIMD-based optimization, and optimization methods for crossword based search as to be presented in Chapter 3, as well as fast LMLA methods as to be presented in Chapter 4.

Table 2.1. A comparison of decoding speed of a LVCSR task running on three different PC workstations with same pruning thresholds and optimization algorithms. Testing set including 333 sentences. Pruning thresholds: bs=240, bw=100, Ns=18,000, Nw=15, bxwd=150, bPLA=245. Memory cost is 196.7 MB. Word error rate is fixed as 9.73%.

[8] See http://www.intel.com/software/products/ipp/.

CPU and Memory	Speed (xRT) (with Fast LMLA and other optimizations)
1.40 GHz+1GB Mem	2.24
3.06 GHz+2GB Mem	1.18
3.20 GHz+3GB Mem	0.98

Chapter 3

System Development of TigerEngine 1.0

In this chapter, the development of a real-time LVCSR decoding engine, which is named TigerEngine[9], is presented. So far the author did all the implementation work for TigerEngine 1.0, resulting in one-pass and one-best real-time decoding for two benchmark tasks: Wall Street Journal 20K and Switchboard 33K.

3.1 Introduction

The basic decoding algorithm for TigerEngine 1.0 is Viterbi time-synchronous beam search which has been described in Chapter 2. Tree copy based search organization as described in Section 2.1.6 is used in TigerEngine 1.0. All knowledge sources including language model, acoustic model and pronunciation dictionary are integrated into a one-pass decoding paradigm. The implementation language is standard C and the compiler is Visual C++. Some highlights of the system development are listed as following:

- The main data structure used for knowledge sources and search management is

[9] This is named after the mascot of the University of Missouri – Columbia (Tiger).

list (“structure array” in C language), which enables direct access of different level of hypotheses and search management facilities (state, phone, word, tree, backtrace array). The sizes of those lists are pre-defined based on the vocabulary size of target task [Ney+ 87, Sixtus 03].

- To reach the state-of-the-art recognition accuracy, cross-word triphone HMM is used, which has been proved to generate significantly higher word accuracy than within-word triphone HMM [Lee+ 89, Hwang + 89, Alleva+ 92, Schwartz+ 92, Sixtus&Ney 02].
- Four heuristic pruning methods and two lookahead techniques as described in Section 2.3.1 are used to reduce search space with little or no loss of word accuracy. Those pruning methods (and the corresponding beam/thresholds) are state-level beam pruning (b_s), word-level beam pruning (b_w), state-level histogram pruning (N_s), word-level histogram pruning (N_w), phone lookahead (b_{PLA}) and crossword language model lookahead (b_{xwd}).
- Trigram language model is integrated into the search as early as possible by using the trigram lookahead instead of using bigram lookahead. The latter method has been used by many reported systems for many years [Steinbiss+ 94, Odell+ 94, Ortmanns+ 96a, Aubert 99], whereas the former method has not been used until recent years [Cardenal+ 02, Slotau+ 02, Li&Zhao 03].
- To deal with the increased search space mainly caused by the use of

crossword triphone models, several techniques are implemented including: recombination after the first phoneme layer, crossword language model lookahead and optimization of the Pronunciation Prefix Tree [Sixtus&Ney 02]. This will be discussed in next two sections.

- To reduce the overhead for trigram language model lookahead, a novel algorithm called **Order-Preserving Language Model Pre-Computing** (OPCP) [Li&Zhao 05] is used, which can significantly speed up the language model lookup and save memory cost without any loss of word accuracy. The details of this algorithm will be presented in next Chapter.

3.2 Crossword Model in Tree-based Search

The search structure used in within-word based tree search was already described in Section 2.1.6. The introduction of crossword models mainly concerns the processing of word boundaries within the search network. In within-word model based search, right-context is not considered for each word's last phone, but in crossword based search, the right context of a word is given by the first phoneme of the successor word. Since the identity of the successor word is unknown at word boundary, all possible monophones will be hypothesized as the right context of current word, which will significantly increase the search space at the word boundaries. The hypothesized right contexts are called **fan-out arcs** [Odell 95, Aubert 99, Beulen+ 99] and the set of fan-out arcs of a word *w* is called

“fan-out arcs of the word *w*” [Sixtus 03]. On the left-hand side of Figure 3.1, the fan-out arcs of word *w* for the right crossword contexts *a*, *b*, *c* and $ are shown (here $ represents a silence model). Each fan-out arc represents a right cross-word context. For each word in the vocabulary, a complete set of fan-out arcs for all possible right crossword contexts has to be generated which increases the number of arcs in the pronunciation prefix tree substantially. Before the search (at the initialization of decoding engine), for each different phoneme pair which can occur at word ends, one set of fan-out arcs for all possible right crossword contexts is generated and stored in a lookup table. During search, if a word end is to be activated whose fan-out arcs are not already contained in the active search network, the instances of those fan-out arcs are created and added dynamically into the active search network [Aubert 99, Sixtus 03].

At the successor tree in the right-hand side of Figure 3.1, the root arc has to be separated into a series of sub-root arcs according to different right contexts of the last word, and the recombination at word-level will be changed from the whole root node in within-word structure to those sub-root arcs and only those word hypotheses with the same left and right contexts can be recombined.

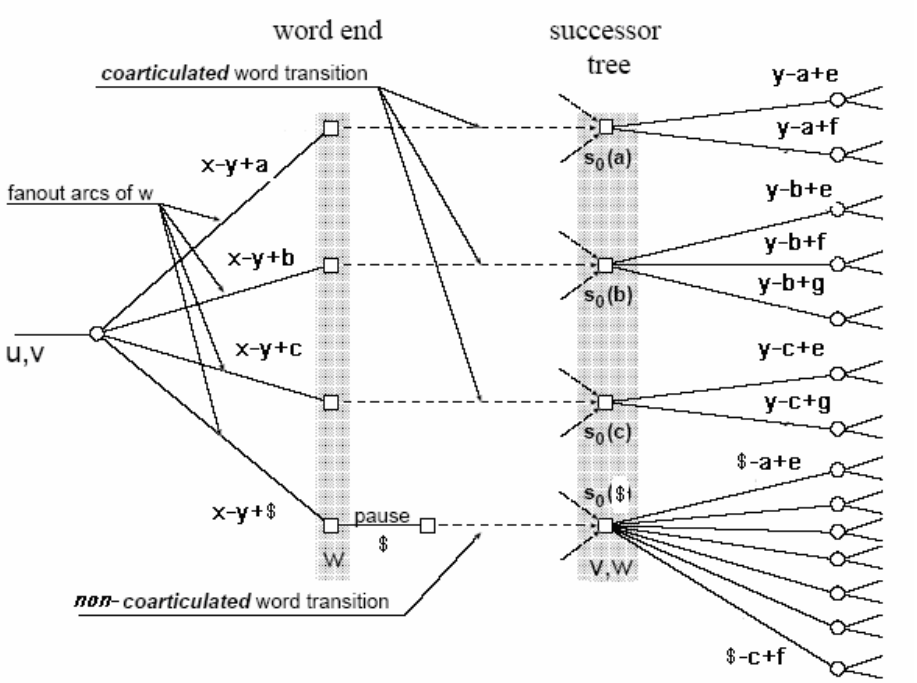

Figure 3.1. Word transition during cross-word search (based on the original figure from [Sixtus 03]).

It should be noted that there is one sub-root arc represented for **non-coarticulated word transition** which is shown at the bottom of Figure 3.1. For this word transition type it is assumed that a pause or silence (represented by the pause arc in Figure 3.1) occurs between the adjacent words which is long enough so that no co-articulation takes place, i.e., the acoustic realizations of the two words do not influence each other. Therefore, in contrast to the **coarticulated word transition** as shown in the upper part of Figure 3.1, the set of phonemes which may follow a non-coarticulated word end is not constrained, and all phoneme arcs occurring in the first layer of the successor tree have to be activated. The right crossword context of the ending word as well as the left crossword context of the

first layer arcs of the successor tree are represented by the special symbol $ in this case. This additional kind of word transition is crucial for good performance in word error rate. It is reported in [Sixtus 03] that not considering this additional word transition results in a performance loss in word error rate of more than 16% relative.

The Viterbi-beam search algorithm for crossword tree based search is similar to the one for within-word as described in Section 2.2, except that the word recursions in formulae (2.8) and (2.9) need to be changed to (3.1) and (3.2), respectively:

$$Q_{vw}(t, s_0(r)) = \max_u \left\{ P(w \mid u, v) \cdot Q_{uv}(t, S_{(w,r)}) \right\}; \tag{3.1}$$

$$B_{vw}(t, s_0(r)) = ind(w, t, Q_{vw}(t, s_0(r)), B_{u_{\max} v}(t, S_{(w,r)})) \tag{3.2}$$

where $s_0(r)$ denotes the sub-root arc according to right-context *r*, and $S_{(w,r)}$ denotes the last HMM state of *w* with right-context *r*.

It should also be noted that in crossword tree based search the **word-level recombination** has to consider the effect of fan-out arc. For example, when bigram language model is used, only those path hypotheses with the same word ending and the same fan-out arc triphone can be merged.

3.3 Methods to Speedup Crossword Based Search

The use of crossword model will significantly increase the search space and a direct implementation by using the same search structure as within-word model results in very slow decoding. Several speedup techniques [Sixtus&Ney 02] are necessary to reduce the

search space with little or no loss of accuracy.

3.3.1 Recombination after the First Phoneme Layer

As discussed in Section 3.2, two types of word transitions need to be considered during crossword tree search: the **coarticulated** word transition and the **non-coarticulated** word transition. Depending on the word transition type generated at the end of a word, different subtrees in the first layer of the successor tree have to be activated via separate root arcs (see Figure 3.1). However, if triphones are used as basic sub-word units, the word transition type generated at the end of a word affects only the HMM of the first phoneme of the successor word, the HMMs of the remaining successor word phonemes are not affected. This allows for the recombination of search paths which have traversed a **coarticulated** word transition with search paths which have traversed the corresponding **non-coarticulated** word transition at the end of the first phoneme layer of the lexical prefix tree. For details of this optimization, refer to [Sixtus&Ney 02].

3.3.2 Crossword Language Model Lookahead

If the search network includes copies of a pronunciation prefix tree, the extension of the accumulated probability of a search path by the language model probability of the final word of the path can not be made until the end of this word is reached. For more efficient pruning many systems take advantage of **language model look-ahead** (see Section 2.3.2),

i.e., the language model probability is distributed over the arcs of the pronunciation prefix tree (e.g., [Steinbiss+ 94, Odell+ 94, Ortmanns+ 96a]). This well known technique can be extended when performing crossword tree search [Ortmanns+ 99, Aubert 99].

As described in Section 3.2, the set of successor words which may follow a word end hypothesis *w* with a specific right crossword context *r* during crossword search is confined to those words whose first phoneme is *r*. In the following, this set of possible successor words is represented by *W*(*r*). The extension and pruning of the HMM states within the fan-out arcs of the words can be conducted more efficiently as follows: when a fan-out arc of word *w* with right crossword context *r* is activated, the accumulated probability of the corresponding partial search path is enhanced by the language model probability of a successor word $\tilde{w} \in W(r)$ which most probably follows *w* [Sixtus 03]:

$$\tilde{Q}_{uv}(t, s_{(w,r)}) = \max_{\tilde{w} \in W(r)} \{P(\tilde{w} \mid w)\} \cdot Q_{uv}(t, s_{(w,r)}) \tag{3.3}$$

where $s_{(w,r)}$ is an HMM state within the fan-out arc of word *w* for right crossword context *r*. By anticipating the language model probability $P(\tilde{w} \mid w)$ for the successor word $\tilde{w}$ of the ending word *w*, the states of the fan-out arcs of *w* can be pruned with a separate beam b_{xwd} which is tighter than the beam used for the conventional state-level beam pruning as discussed in Section 2.3.1. By using this **cross-word language model look-ahead pruning**, the number of HMM states in fan-out arcs can be reduced considerably which results in a rather significant acceleration of the search.

To minimize the computational overhead during search, for each word *w*

and context r the anticipated language model probability $\max_{\tilde{w}\in W(r)}\{P(\tilde{w}\,|\,w)\}$ is computed beforehand and stored in a look-up table. In order to keep the memory cost as low as possible, a bigram language model instead of a trigram language model is used.

3.3.3 Optimization of the Pronunciation Prefix Tree

3.3.3.1 Tying of Fan-Out Arcs

As described in Section 3.2, the search space of crossword based structure will be much larger than within-word based structure mainly because of the fan-out arcs at the word boundaries. The number of fan-out arcs can be largely compressed by exploiting the tying information obtained in the acoustic model training, as mentioned in Section 2.1.2. Those different fan-out arcs (different logical triphones) from the same parent arc but sharing a same physical triphone model can be merged into one fan-out arc. For example, in WSJ 20K task, by this merging, the number of fan-out arcs are reduced by 23.7% from 884,604 into 675,400.

One side-effect of this fan-out arc tying is that one fan-out arc may be generated with more than one right contexts. All those different right-contexts will be extended in the sub-root of successor tree to present different first-layer triphones with the un-tied right-contexts as their mid-phones. This brings a problem about how to efficiently represent those fan-out arcs with tied right-contexts. As used in [Sixtus&Ney 02], the

different sets of right contexts are represented by additional context indices which are treated like normal right contexts during search. For the lexicon considered in Table 3.1, this leads to 419 right context indices that represent the different sets of right cross-word contexts including the 44 original sets containing only one right context. Table 3.1 lists the arc numbers before and after this fanout arc tying for this task.

Table 3.1. Statistics of the lexical prefix tree structure for the WSJ 20K task with and without merging of triphone arcs which share the same HMM state sequence due to state tying. 7,771 tied states were used for this tying.

# vocabulary word		19,982
# lexical entries		21,945
# lexical tree layers		19
# within-word arcs (without fan-out arcs)		201,449
# within-word arcs after compression (# LM arcs)		52,147
# fan-out arcs	without merging	884,604
	with merging	675,400
# different right contexts	without merging	44
	with merging	419

3.3.3.2 Compression of Pronunciation Prefix Tree for Language Model Lookahead

Another optimization for lexical prefix tree is to compress the lexical tree used for language model lookahead [Ortmanns+ 96a]. This is based on the fact that for those lexical arcs with single child, its language model lookahead (LMLA) score will be the same as its child arc, so the parent arc can be merged with its child arc. There are many

single-child paths in a pronunciation prefix tree for a LVCSR task, so this compression will bring much saving for the LM lookup during the search. Table 3.1 has already listed the number of arcs for the compressed tree for WSJ 20K task, i.e., 52,147, which is only about 1/4 of the size of the original lexical tree. It should be noted that this compressed tree is only used for language model lookup, and the original large lexical tree is still needed for the search. This compressed tree is also called **Language Model tree**.

3.4 Test Tasks

Two standard tasks with different vocabulary sizes and speech styles were evaluated on TigerEngine 1.0. Table 3.2 gives detailed information about the two tasks. The first one was Wall Street Journal (WSJ) with a 20K vocabulary. The training set was the standard WSJ0 and WSJ1 corpora provided by LDC[10] and it included 37,511 utterances and 284 speakers of short-term Speaker-Independent continuous speech data set. The sampling frequency was 16 KHz, and the total training data was about 81.5 hours long. The acoustic model was trained by using HTK toolkit [Young+ 00]. A phone set consisted of 44 monophones was defined based on the pronunciation dictionary released from LDC[11]. For within-word models, there are 4,902 HMM tied states with 16 Gaussian mixture components in each state, and 7,909 tied triphones and 9,970 untied triphone. For cross-word models, there were 7,771 HMM tied states with 16 Gaussian mixture

[10] See http://www.ldc.upenn.edu/.
[11] See LDC website.

components in each state, and 38,812 tied triphones with separate modeling for both cross-word triphones and within-word triphones. The total number of position-dependent (PD) crossword triphones was 151,748. Speech feature vector represented in every time frame was 39-dimensional, including 13 MFCC parameters plus their first- and second-time derivatives. Speech analysis was processed by a frame shift of 10 ms and window length of 25 ms. Evaluation data was one of standard WSJ evaluation testing sets: SI_ET_20 in Non-Verbalized-Punctuation (NVP) part. There were totally 333 testing utterances from 8 speakers. The language model was standard Nov 92 testing LM provided by LDC, including 19,982 unigrams, 3,518,595 bigrams, and 3,153,527 trigrams. Recognition word error rate on this task was about 9.2%~9.9%, which reached the state-of-the-art level for this task [Reichl&Chou 00].

Table 3.2 Detailed information for the two testing task of TigerEngine 1.0.

Task	**Wall Street Journal 20K**	**Switchboard 33K**
Speech Style/Environment	Read/Clean	Spontaneous/Telephone
Sampling Rate	16 KHz	8 KHz
Training Data	WSJ0+WSJ1	Switchboard-I(2995 conversation)+ MS'98 Switchboard-I
Training Data Size	81.5 hours	270 hours
#Speakers	284	> 1000
Feature type	MFCC_13+1st+2nd derivatives (39-dimension)	MFCC_12+Energy+1st+2nd derivatives (39-dimension)
Other Front-end Processing	Cepstral Mean Subtraction (CMN)	CMN+Side based Variance Normalization
AM type	PD (Position-dependent) Crossword Triphone	Non-PD Crossword Triphone

#Tied State	7,771	5,887
#Tied triphone	38,812	18,836
#Logical triphones	151,748	77,660
#N-grams(1/2/3)	19,982/3,518,595/3,153,527	33,216/3,246,315/9,966,270
Testing Data	WSJ Nov92-NVP Testing Set (333 sentences)	Switchboard-I section of Hub5e 2001 Testing Set (Female part, 776 sentences)
Best Word Error Rate	9.18%	42.11%

Table 3.3 lists the acoustic model and recognition performance of within-word model and crossword model in WSJ 20K task. Conservative pruning thresholds were used in each testing to obtain the lowest word error rate. Same training data, mixture number per state and 333 testing sentence were used. It can be seen that the crossword model can bring as much as 12.3% word error rate reduction, which is a significant improvement. On the other hand, crossword model based search needs about 41 MB more memory space, and 3 xRT slower than within-word model based search.

Table 3.3. Comparison of within-in word model and crossword model in WSJ 20K task. Pruning thresholds for within-word model: b_s=320, b_w=120, N_s=45,000, N_w=30, b_{PLA}=255; for cross-word model: b_s=320, b_w=120, N_s=60,000, N_w=30, b_{PLA}=255, b_{xwd}=150.

	Within-word	Crossword
# logical triphone	9,970	151,748
# tied triphone	7,909	38,812
# tied state	4,902	7,771
Word Error Rate (%)	10.47	9.18
Error Reduction Rate		12.3%
Speed (xRT)	2.82	5.93
Peak Memory Cost (MB)	166.1	207.6

The second task, Switchboard (SWB), was much more difficult than the WSJ task, because SWB was telephone spontaneous speech instead of clean read speech as in WSJ. A set of gender-dependent crossword triphone model was obtained from ISIP of Mississippi State University [Sundaram+ 01]. The training data included 60 hours of Switchboard-I data (2,995 conversation sides) for base training and 210 hours of MS'98 Switchboard-I data for 16-mixture training [Sundaram+ 01]. Sampling frequency was 8 KHz and speech was analyzed with a window length of 25 ms and a frame shift of 10 ms. Speech feature vector consisted of 12 MFCCs, 1 log energy, and their first-order and second-order time derivatives. Side-based cepstral mean subtraction and variance normalization were employed. The female part of the first 20 conversations (Switchboard-I) of Hub5 2001 English evaluation set [Martin&Przybocki 01] was taken as the test set. Utterance segmentation (beginning and end time) was provided by NIST[12], and the segmented utterance set consisted of 776 sentences. Correspondingly, the female crossword triphone models were used which included 5,887 tied HMM states with 16 Gaussian mixtures per state, and 18,836 tied HMM models. Position-dependency was not used in acoustic modeling, which resulted in totally 77,660 crossword triphones based on a similar set of 44 monophones. A large language model was provided by SRI [Stolcke+ 01], which included 33,216 unigrams, 3,246,315 bigrams, and 9,966,270 trigrams. Also used in the developing test was a simplified LM including 319,938 bigrams and 137,938 trigrams, which was

[12] See http://www.nist.gov/speech/tests/ctr/h5_2001/index.htm.

pruned from the large LM by using SRI LM toolkit [Stolcke 02]. In current work, certain additional acoustic processing techniques such as HDLA, VTLN and speaker adaptation [Woodland+02] were not used. Recognition word error rate on this task was about 43~45%. More experiments about this task will be reported in next Chapter as the fast LM lookup method will be used to reach real-time speed performance.

Chapter 4

Fast and Memory-Efficient Language Model Lookup

4.1 Introduction

As described in Section 2.3.2, N-gram Language Model (LM) lookup is an important component in the state-of-art Large Vocabulary Continuous Speech Recognition (LVCSR) systems based on dynamic search network. In lexical-tree based LVCSR, LM lookup includes exact N-gram lookup at tree leave nodes[13] where the last word's identity of a current search path is known, as well as LM lookup at the tree internal nodes, referred to as LM look-ahead (LMLA) [Ortmanns+ 96a]. At an internal node, because the last word's identity is unknown, approximate LM score, called LMLA score, is used to reduce decoding efforts. Given a LM context, LMLA score is usually taken as the maximal value

[13] In this Chapter, lexical tree "nodes" will be used to represent tree "arcs".

of the N-gram scores of those words that can be reached by the current lexical node. By using LMLA, LM knowledge can be integrated with acoustic model knowledge as early as possible during speech decoding, resulting in faster speed and higher accuracy [Ortmanns+ 96a]. On the other hand, compared with LM lookups at tree leave nodes only, LMLA has a significant computational overhead. For each internal tree node, it needs to lookup the LM scores of all those reachable words and maximize over their scores [Deshmukh+ 99], or in tree-copy based search organization [Ney&Ortmanns 99], a dynamic programming procedure needs to be used to fill the LMLA scores from the tree leave nodes to the first-layer nodes for each new LM context. To save time in LMLA, many one-pass decoding systems use lower-order N-gram look-ahead scores to approximate N-gram look-ahead [Ortmanns+96, Ortmanns+ 97, Ney+ 98, Aubert 99]. However, some recent work have shown that using trigram rather than bigram or unigram LM look-ahead can result in faster speed and higher accuracy [Slotau+ 02, Cardenal+ 02].

In LVCSR search, the problem of LM lookup for N-grams can be posed as the following: given an N-word string, $w_{n-N+1},...,w_{n-1},w_n$, how to efficiently access the corresponding N-gram scores $P(w_n \mid w_{n-N+1},...,w_{n-1})$? The N-1 word history $w_{n-N+1},...,w_{n-1}$ is called the **LM context** for an N-gram LM. Since the number of LM lookups is huge even for a short utterance of several seconds, it is necessary to cut down LM lookup time as much as possible. There are different ways to access N-gram scores, with different costs in memory and computation time, for example, binary

search, non-prefect hashing, or perfect hashing. The first two methods have been used for many years, whereas minimum perfect hashing (MPH)[14] was first proposed in 2002 by two independent efforts [Cardenal+ 02, Zhang&Zhao 02], taking advantages of algorithmic developments in MPH Functions (MPHFs) [Tarjan 83, Fox+ 92, Czech+ 92, Jenkins 95]. In [Zhang&Zhao 02], experimental results of a Switchboard task showed significant advantage of a MPH based LM lookup method in both memory and computation over a binary search method [Deshmukh+ 99] as well as a non-perfect hashing method.

In LVCSR systems, the time and memory usage of LM lookup is heavily dependent on data structure and process flow. For example, in order to further speed up MPH-based LM lookup, several LM cache structures were proposed. In [Li &Zhao 03], the LMLA scores of different LM contexts were stored in a set of subtrees dissected from the original lexical tree. In [Cardenal+ 02], the LMLA scores of different LM contexts were stored in each lexical node. Both methods proved effective in reducing LMLA time, with the node-based cache method taking less memory than the subtree-based method. Clearly, it is desirable to maximize the speed of LM lookup while minimizing the memory cost. In most cases, the two aspects contradict, where faster LM lookup is achieved at the cost of higher memory expense. LM Context Pre-computing (LMCP) [Cardenal+ 02] is such an example.

In LMCP method, each time a new LM context (or tree copy) appears in the search, LM probabilities for the entire vocabulary words with this context are retrieved into an

[14] Minimum Perfect Hashing (MPH) is a type of perfect hashing with minimal size of hashing table, that is, the size of hashing table is equal to the number of keys.

array. In subsequent search steps involving this LM context, LMLA score for each lexical node is then easily obtained by a maximization over the scores of those words that can be reached from this node, and LM lookup at a leaf-node becomes a direct array reference by the last word. The maximization operation can be made fast if every node keeps a list of indices of its leaf-node words[15]. Because hashings for LM scores are performed in a batch for each LM context, better cache locality and hence speed are achieved. In addition, batch processing of hashing operations is efficient due to shared word history in a LM context, where 50% reduction of hashing time was reported in [Cardenal+ 02]. The overall computation cost of LMCP is proportional to the product of vocabulary size and average number of new LM contexts per time frame. In [Cardenal+ 02], with a vocabulary of 20K and only one new LM context appearing in every other three time frames, LMCP further speeded up LMLA on top of the node-based LM cache method. In our experiments on the WSJ 20K and Switchboard 33K tasks to be reported in Section 5, however, LMCP improves MPH-only based LMLA, but slows down LM cache based LMLA, because the average new LM contexts per frame in these two tasks are much larger than that in [Cardenal+ 02].

A shortcoming of LMCP is that the storing of pre-computed LM probabilities incurs a memory cost that is proportional to the product of the vocabulary size and the maximum

[15] This record of word indices at each lexical node only takes up a small memory space because a compressed lexical tree is used for LMLA with much fewer number of nodes compared with the original lexical tree. The compressed lexical tree means only those lexical nodes with more than one child nodes are kept, because those nodes with one child node will share the LM look-ahead score with the child node [Ortmanns+ 96a].

number of active LM contexts in each frame. For example, the maximum number of active LM contexts per frame is 104 for the WSJ 20K task, and the memory cost for storing trigram scores of occurred LM contexts is 7.9 Mega Bytes (MB).

By integrating the ideas of LM context pre-computing and order-preserving MPH based LM lookup [Li &Zhao 03], a novel Order-Preserving LM Context Pre-computing (OPCP) method is proposed in this dissertation for fast and memory-efficient LM lookup. In OPCP, assuming that the last words of N-grams for each fixed LM context are sorted by the order of vocabulary word list, e.g., alphabetically, we then need to do hashing only once to locate the first N-gram for a given LM context, and the rest N-grams in this context can be obtained sequentially without hashing. On one hand, this saves significant computation in LMCP because the number of hashing operations is reduced from the size of vocabulary to one for each LM context. On the other hand, the number of hashing keys is reduced from the number of N-grams to the number of distinct word histories in N-gram, and a fast integer-key based MPHF can therefore be used in place of slow string-key based MPHF [Zhang&Zhao 02]). In LMCP or MPH-only methods, if the number of N-grams reaches several millions, which is typical in LVCSR applications, then only string-key based MPHF can be obtained. In the proposed OPCP, however, since the number of distinct N-gram contexts (the number of keys of MPHF) is only 20~25% of original N-gram size, integer-key based MPHF can be obtained in general.

OPCP also brings significant memory savings over MPH-only or LM cache

methods. This benefit results from the following four factors. Firstly, order-preserving building of LM context[16] reduces memory usage for the storage of original LM. For each N-gram, only the last-word index needs to be stored in OPCP instead of N word indices as in LMCP or MPH-only methods. Secondly, in LMCP or MPH-only methods, besides the MPH table for trigrams, a MPH table is also needed for bigrams as back-off LM. In contrast, in OPCP there is no need for the bigram MPH table. The number of different contexts for bigram is simply the vocabulary size, and an array is sufficient for order-preserving building of a bigram context. This brings both memory and computation savings over LMCP. Thirdly, the above described simplification to MPH for trigram also reduces memory cost because the hashing table of integer-key based MPHF is more compact than that of string-key based MPHF [Zhang&Zhao 02]). Fourthly, in MPH or LMCP methods, node-based cache or tree-based cache is usually used to speedup LM lookup, at the cost of increased memory usage. In OPCP, however, LM cache is no longer needed because the LM lookup procedure is in fact faster than accessing node-based or tree-based caches[17]. Overall, these advantages of OPCP result in a significant reduction of memory cost, which makes OPCP much more efficient in both time and space than MPH-only, LM cache, or LMCP methods.

This Chapter is organized as follows. In Section 4.2, the MPH-based LM lookup

[16] In this chapter, "LM context building" will be used interchangeably with "LM context pre-computing".
[17] The storage of an array of N-gram scores for a LM context can also be considered as a kind of LM cache. We distinguish this from conventional LM cache because the scores stored in the LM context are "raw" LM probabilities (without maximization) instead of LMLA scores (after maximization).

methods are introduced. In Section 4.3, the node-based LM cache and the LMCP methods are described. In Section 4.4, the proposed OPCP method is presented. In Section 4.5, experimental results based on TigerEngine 1.0 are reported. Finally, conclusions are made in Section 4.6.

4.2 Minimum Perfect Hashing (MPH) based LM Lookup

Two independent efforts were first reported by [Cardenal+ 02] and [Zhang&Zhao 02]) on using Minimum Perfect Hashing to speed up LM look-ahead in LVCSR. There are several differences between the two efforts. Firstly, different hashing functions were used. In [Cardenal+ 02], the MPH functions were mainly based on modulo operation whereas in [Zhang&Zhao 02]) bitwise logical operations and bit shift operations were used. Secondly, only string-key based MPH function (MPHF) was studied in [Cardenal+ 02] whereas both integer-key and string-key based functions were investigated in [Zhang&Zhao 02]). Thirdly, the MPHFs generated in [Cardenal+ 02] are order-preserving, but those generated in [Zhang&Zhao 02]) have no such property. The three different MPH functions for trigram and bigram access as used in these two efforts are briefly described below.

4.2.1 String-key based Order-Preserving MPHF

Given m trigrams, $(w_{i0}, w_{i1}, w_{i2}), i = 0, 1, ..., m-1,$ a string key is formed for each trigram as $K_i = (c_{i1}c_{i2}...c_{il})$, where $l = 2 + \sum_{k=0}^{2} strlen(w_{ik})$, with 2 accounting for two blanks

between three words. The indices of hash table entries are given by $h(K_i)$ and are generated by the following operations:

$$v_1 = \sum_{j=1}^{l} Tab\_a(j, c_{ij}), \tag{4.1}$$

$$v_2 = \sum_{j=1}^{l} Tab\_b(j, c_{ij}), \tag{4.2}$$

$$h(K_i) = (g[v_1 \bmod n] + g[v_2 \bmod n]) \bmod m. \tag{4.3}$$

where $g(\cdot)$ is an integer table of n ($>m$) entries, $Tab\_a(\cdot,\cdot)$ and $Tab\_b(\cdot,\cdot)$ are 2-dimensional integer tables of size L $\times$ P, with L the maximal key length and P the total number of ASCII codes, i.e., 256.

Tables $Tab\_a(\cdot,\cdot)$, $Tab\_b(\cdot,\cdot)$ and $g(\cdot)$ are obtained by an iterative training algorithm of [Czech+ 92], and the expected number of iterations for convergence is dependent on the value of n. Reasonable convergence time can be achieved for $n \geq 2m$ [Czech+ 92]. In our implementation, $n = 2m$ was used.

The string-key based MPHF generated in this way is an order-preserving MPHF (OP-SMPHF), which means $h(K_i) = i$, that is, trigram string keys K_i are sequentially mapped into hash table entries i. This property can be exploited to speed up a batch of LM lookup operations as demonstrated in [Li&Zhao 03].

4.2.2 String-key based MPHF (non-Order-Preserving)

In [Zhang&Zhao 02]), the string-key MPHF is implemented as the following,

$$h(K_i) = (v_1 \,\&\, A) \wedge tab(v_2 \,\&\, B) \tag{4.4}$$

where v_1 and v_2 are the same as in (1) and (2), A and B are two 32-bit constants, $tab(\cdot)$ is a mapping array, "^" and "&" are bit-wise XOR and AND operations, respectively. The size of $tab(\cdot)$ is $B+1$, with $B+1=2^d < m$ and d is a constant number. The algorithm described in [Tarjan 83, Fox+ 92, Jenkins 95] is used to search for A and B as well as the array $tab(\cdot)$.

The string-key based MPHF thus generated is not order-preserving, and is referred to as non-OP SMPHF or simply SMPHF, to be distinguished from OP-SMPHF. Because the hashing operation in SMPHF only involves bitwise operations such as XOR and AND, it is in general faster than the OP-SMPHF. Furthermore, the size of $tab(\cdot)$, i.e., $B+1$, is always less than the number of keys m, whereas in OP-SMPHF, the size of $g(\cdot)$ is twice the number of keys. Therefore the memory cost of SMPHF is only about half of OP-SMPHF's.

The two string-key based MPHFs, OP or non-OP, can also be used for bigram MPH functions, where a key string is formed by two words connected by one blank.

4.2.3 Integer-key based MPHF

Integer-key based MPHFs are separately designed for bigrams and trigrams [Zhang&Zhao 02]). Word indices are represented in short integers of 16 bits with the underlying assumption that the vocabulary size is less than 64K.

For bigram $(w_{i0}, w_{i1}), i=0,1,...,m-1$, a 32-bit key is formed by placing w_{i1} in the high 16-bits and w_{i0} in the low 16-bits, and the resulting number is denoted as K_i

$= ((w_{i1} << 16) + w_{i0}) + S$, where S is a 32-bit random number. The MPHF is obtained as following:

$$\begin{aligned} key &= K_i; \\ key &\wedge= (key >> 16); \\ key &+= (key << 8); \\ key &\wedge= (key >> 4), \end{aligned} \tag{4.5}$$

$$h(K_i) = x(key) \wedge tab[y(key)]. \tag{4.6}$$

where $x(\cdot)$ and $y(\cdot)$ are two functions involving only bit-wise operations, $tab(\cdot)$ is a mapping array as in SMPHF. In (4.5), three bit-mixing operations are performed on each integer key K_i to obtain unique values of $x(key)$ and $y(key)$ for each $i = 0, 1, ..., m-1$. The algorithm described in [Tarjan 83, Fox+ 92, Jenkins 95] is used to search for $x(\cdot)$, $y(\cdot)$ and $tab(\cdot)$.

For trigram $(w_{i0}, w_{i1}, w_{i2}), i = 0, 1, ..., m-1$, a key consists of three word indices. Since concatenating three 16-bit integers requires 48 bits which exceeds the 32-bit integer representation in commonly used machines, two 32-bit intermediate integers X and Y are first computed for each trigram, where $X = ((w_{i2} << 16) + w_{i1})$ like the bigram, and Y is generated by adding to w_{i0} a 32-bit random number, i.e., Y = w_{i0} +*S*. A more complex bit-mixing procedure is performed to make the bits of X and Y interact. The MPHF is the same as (4.6), and a similar algorithm is used for the search of $x(\cdot)$, $y(\cdot)$ and $tab(\cdot)$. Please refer to [Tarjan 83, Fox+ 92, Jenkins 95] for the detailed search procedure.

The MPHF thus generated is not order-preserving. In subsequent discussions, the integer-key based MPHF is abbreviated as IMPHF.

Both the string-key and the integer-key based MPHFs have been shown successful for fast access of bigram and trigram LM scores. In general, string-key based MPHF is more flexible than integer-key based MPHF since it is readily applicable to N-grams with N greater than 3. In the case of integer keys, the higher is the order N of N-gram, the less straightforward it becomes to find the functions $x(\cdot)$ and $y(\cdot)$ to generate distinct $(x(key), y(key))$'s with the 32-bit constraint. Even for lower order N-grams, if the number of keys is large, then it becomes difficult to obtain integer-key based MPHF in an acceptable time[18].

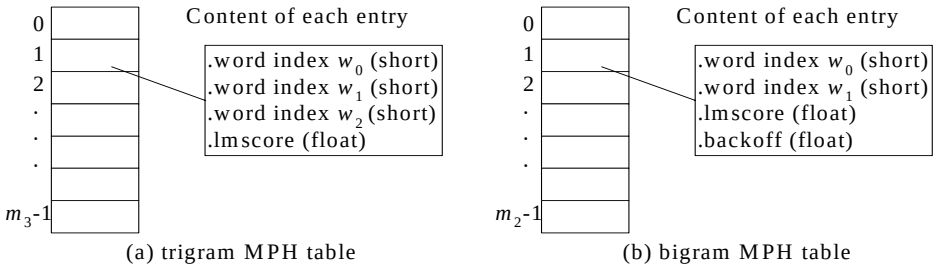

Figure 4.1. Data structure of MPH tables used for (a) trigram and (b) bigram (m_3 = size of trigram, m_2 = size of bigram)

Table 4.1. Memory cost (Bytes) of different MPHFs used for trigram (m_3 = the size of trigram, d is a constant determined in MPHF training algorithms, l_3 = the maximal length in characters of a trigram string)

MPHF	trigram storage	*tab* or *g*	*Tab_a* & *Tab_b*	Overall
IMPHF	$10* m_3$	$4*2^d < 4*m_3$	0	$< 14* m_3$
SMPHF	$10* m_3$	$4*2^d < 4*m_3$	$2*4*256*l_3$	$< 14* m_3+2048*l_3$
OP-SMPHF	$10* m_3$	$8* m_3$	$2*4*256*l_3$	$18* m_3+2048*l_3$

[18] In Intel SSE2 instruction set or IA-64 architecture, it is possible to use 64-bit integer, and so that IMPHF will be available for higher order N-grams and large-size N-gram list. Actually, it is possible to use 64-bit integer data type in Visual C++ 6.0 (__int64).

Table 4.2. Memory cost (Bytes) of different MPHFs used for bigram (m_2 = the size of bigram, d is a constant determined in MPHF training algorithms, l_2 = the maximal length in characters of a bigram string)

MPHF	bigram storage	*tab* or *g*	*Tab_a* & *Tab_b*	Overall
IMPHF	$12*m_2$	$4*2^d < 4*m_2$	0	$< 16*m_2$
SMPHF	$12*m_2$	$4*2^d < 4*m_2$	$2*4*256*l_2$	$< 16*m_2+2048*l_2$
OP-SMPHF	$12*m_2$	$8*m_2$	$2*4*256*l_2$	$20*m_2+2048*l_2$

Figure 4.1(a) shows the data structure of MPH table for trigrams. Table 4.1 gives the memory costs in different MPHFs for trigram MPH tables, where m_3 is the size of trigram and l_3 is the maximal length in characters of a trigram string. ($l_3=3*l_1+2$, where l_1 is the length of the longest word in the vocabulary.) Each entry of the MPH table includes 4 elements, i.e., three word indices w_0, w_1, w_2, and one LM score. Each word index uses 2 bytes for vocabulary size less than 64K, and a float-type LM score uses 4 bytes. In the lookup tables $tab(\cdot)$, $g(\cdot)$, $Tab\_a(\cdot,\cdot)$ and $Tab\_b(\cdot,\cdot)$, each entry is a 4-bytes integer. As can be seen, IMPHF consumes the smallest amount of memory, SMPHF takes comparable or more memory in comparison with IMPHF, and OP-SMPHF takes much more memory due to the requirement by table $g(\cdot)$ on $n = 2m_3$.

The data structure for bigram MPH table is shown in Figure 4.1(b). Table 4.2 gives a comparison for the bigram MPH tables, where m_2 is the size of bigram and l_2 is the maximal length in characters of bigram string. ($l_2=2*l_1+1$, where l_1 is the length of the longest word in the vocabulary.) Different from the trigram table, one more float data is

needed in each entry to store bigram back-off score, according to the commonly used back-off formula:

$$P(w_2 \mid w_0, w_1) \approx P(w_2 \mid w_1) \cdot \alpha(w_0, w_1) \tag{4.7}$$

where $\alpha(w_0, w_1)$ is the back-off coefficient for trigram (w_0, w_1, w_2).

The speed and memory performances of different MPHFs in LM lookup for LVCSR decoding will be reported in Section 4.5, and comparisons will be made with conventional binary search and non-perfect hashing.

4.3 LM Cache and LM Context Pre-computing (LMCP)

Although the above described MPH-based method of LM lookup is significantly faster than binary search or non-perfect hashing, it still takes a significant share of decoding time in LVCSR systems. Recently, more systems started using trigram instead of bigram or unigram LMLA [Slotau+ 02, Cardenal+ 02] for higher word accuracy, and therefore more efficient methods are desired to further speedup LMLA. LM cache and LM Context Pre-computing (LMCP) offer such options.

4.3.1 LM Cache

LM cache is motivated by the observation that for many lexical nodes in a tree-structured search network, LM lookups are repeated many times in a short time period, and therefore using a LM cache to store recently accessed LMLA scores will save time for future

repeated lookups. Several LM cache scenarios were investigated and they are summarized here.

The first cache scenario is to cache the LM look-ahead scores in a whole, compressed lexical tree for each active LM contexts and keep multiple tree copies for several active LM contexts [Ortmanns+ 96a, Ney&Ortmanns 99]. Two indices are needed to locate a cache entry: one for LM contexts (e.g., for a trigram (u, v, w), the context index is obtained from its two-word history as u*V+v, where V is the vocabulary size), and the other for node index in the lexical tree. In general, accessing a cache entry by two indices is slow, and keeping LMLA scores of a LM context for all lexical nodes in one cache structure requires large memory space.

The second cache scenario is subtree-based cache as in [Li&Zhao 03], where the compressed lexical tree is split into a set of subtrees that are rooted in the first-layer nodes of the whole tree, and cache access is based on three indices: the LM context, the subtree index, and the node index in the subtree. Although speed is also slow due to the use of three indices to locate one cache entry, there are two advantages in this method. Firstly, because the subtrees are rooted in the first-layer nodes and phone look-ahead [Ortmanns+ 97a] may prune away many first-layer nodes, the total number of active subtree caches is relatively small, and thus the memory demand for LMLA score storage is small in comparison with the first cache scenario. Secondly, in a subtree cache, LMLA scores can be filled efficiently by Order-Preserving MPHF (OP-SMPHF) that reduces hashing time for a batch of LM

score lookups [Li&Zhao 03]. Yet, it still costs considerable memory space to store the tables of OP-SMPHFs as well as the LMLA scores in subtree caches.

The last cache scenario is node-based cache as used in [Cardenal+ 02], where each lexical node is associated with a LM cache, and the access of cache entry needs only one index, i.e., LM context. Cache size can be kept very small because only a few active LM contexts may exist in each node, and so the cache access speed is fast. The memory cost for node-based LM cache is the smallest compared with tree-based or subtree-based cache structures, since no redundant LM scores are kept for non-active nodes as in the previous two cache scenarios. Therefore, only node-based LM cache is exploited in our current decoder and experiments.

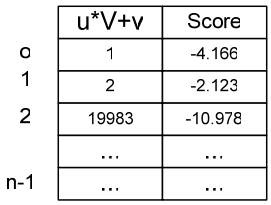

Figure 4.2. Node-based LM cache, where n is the cache size

Figure 4.2 shows node-based LM cache used in our decoding system. Each cache entry includes two members: the LM context index (u*V+v), and the trigram LMLA scores for this LM context at the current lexical node. The size n of LM cache is empirically set for a good trade-off between speed and memory[19]. Random replacement policy is used for updating LMLA scores in each cache. The total number of LM cache is equal to the number

[19] We first optimized the cache size with respect to LMLA time, and then reduced the cache size while keeping the LM lookup time at the optimized level. The optimization was performed on a development data set.

of nodes in the compressed lexical tree.

4.3.2 LM Context Pre-computing (LMCP)

In [Cardenal+ 02], LM Context Pre-computing or LMCP was proposed to further speedup LM lookup in addition to node-based LM cache. Given a new LM context, LMCP pre-computes LM probabilities for all words in the vocabulary and stores them in an array. Then, for any lexical node in this LM context, the LMLA score is obtained by maximizing over the subset of LM scores in this array, which corresponds to the words that are leave nodes of the current lexical node. The maximization is carried out very efficiently as the subset of the word indices is pre-stored in each lexical node.

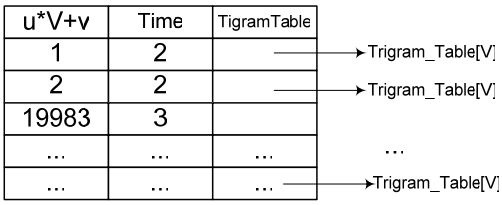

Figure 4.3. Trigram context array used in LMCP

Figure 4.3 shows the data structure of trigram context array used in LMCP. Every entry includes three elements: index of LM context (u*V+v), activation timestamp for this entry, and pointer to the trigram score table of that context. Each trigram table is a float-type array with the size of vocabulary V. The activation timestamp is used for replacing non-active entries in the context array.

Figure 4.4 illustrates the two-step procedure of LM context pre-computing in

LMCP. In Step 1, when a LM context (u, v) is generated, it looks up the trigram context array to see if the context already exists. If yes, stop; else, go to the next step. In Step 2, it enters a loop to retrieve trigram scores for all words in the vocabulary by performing trigram hashing V times. If any trigram (u, v, w) does not exist, it will enter the backing-off procedure to get the bigram score for (v, w) and the back-off coefficient for (u, v) through bigram hashing. If the bigram score does not exist, it will look up unigram array for unigram backing-off.

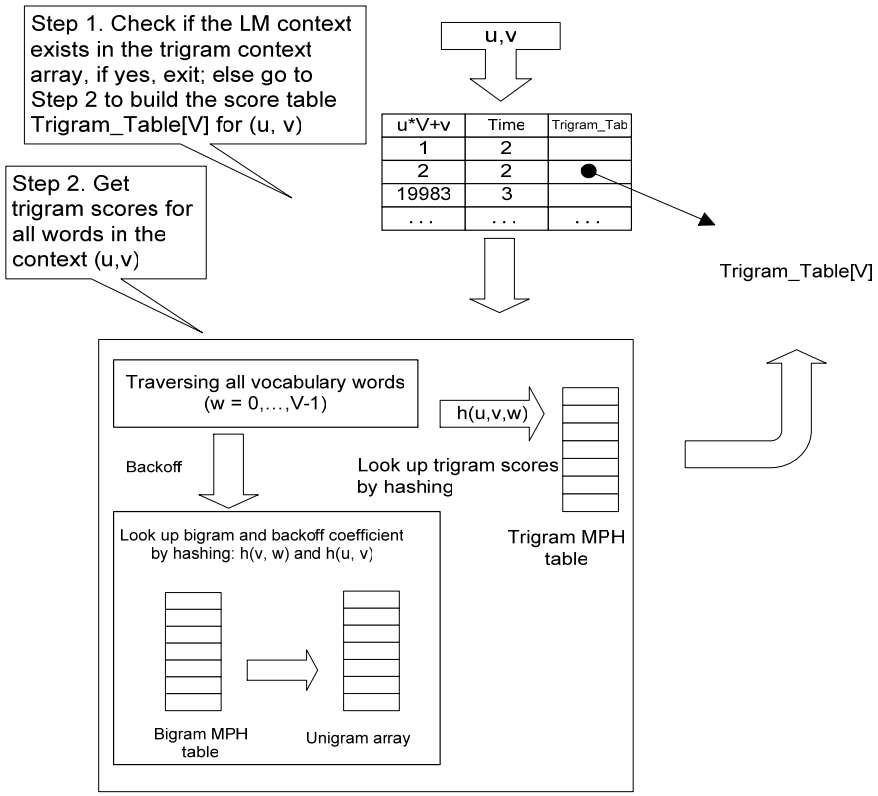

Figure 4.4. The procedure of LM context pre-computing in LMCP

Figure 4.5 illustrates the LM lookup procedure in LMCP. The array of trigram

scores for LM context (u, v) can be accessed by an entry index of LM context array stored in the tree copy of (u, v) or in each active node of this LM context for a single-tree based search organization [Alleva 97]. LM lookups for LMLA score at an internal node and for regular LM score at a leaf node are shown. The LMLA score for an internal node S is obtained by maximizing over the trigram scores for the words that are leaf nodes of this lexical node. The LM score at a leaf node corresponding to word w is obtained from the trigram score array by a direct reference using word index w.

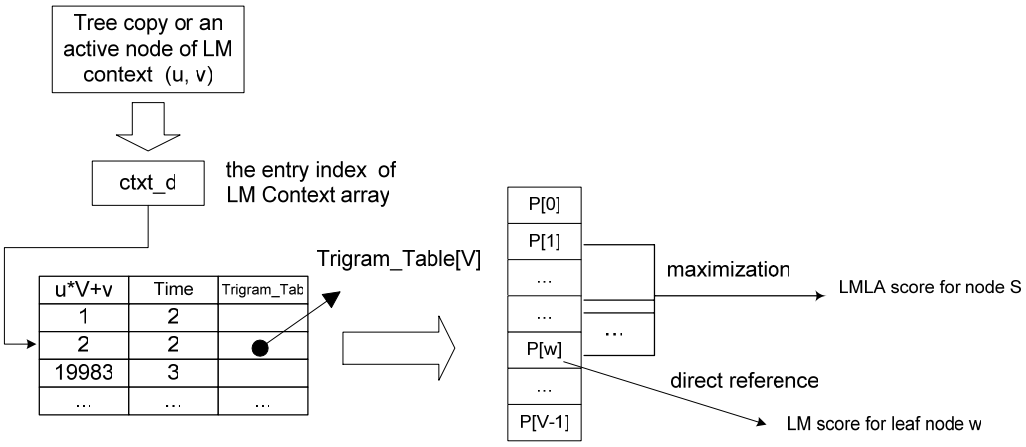

Figure 4.5. LM lookups in LMCP

One advantage of LMCP is that the history words of all N-grams in a fixed LM context are identical, and so the trigram hashing formulas of string-key based MPHF as (4.1) and (4.2) can be simplified by computing only the key value for the last word (note that there is no such advantage for IMPHF). Another advantage of LMCP is that LM lookup at a word-ending or tree leaf node is reduced to a direct reference by word index. In contrast, in

MPH-only or MPH+Cache methods, LM lookup at leave nodes still need hashing operation.

LMCP has the following problems. Firstly, not all the LM probabilities stored in a LM context array will be used in the search, and a proportion of hashing operations for the entire vocabulary is wasted. The waste becomes significant when the average number of new LM contexts per frame is large. Secondly, additional memory space is needed to store the LM scores from LM context building, which is proportional to the product of vocabulary size and the maximum number of active contexts in a time frame. Experimental results on LMCP will be given in Section 4.5 and will be compared with the proposed OPCP method.

4.4 Order-Preserving LM Context Pre-computing (OPCP)

In LMCP, pre-computing LM scores for each active LM context helps reducing computation in LMLA at each lexical node, but a proportion of the hashing operations for the entire vocabulary words are unnecessary. If this overhead can be reduced, then LM lookup may become much more efficient. Combining this viewpoint with our previous experience in exploiting order-preserving method to speed up LM lookups [Li&Zhao 03], the Order-Preserving LM Context Pre-computing (OPCP) method is developed. Observe that in a given LM context, word histories are the same for all N-grams, and the N-gram

scores can be stored according to the order of word indices in the original N-gram list[20]. Thus if for a given LM context the position of the first LM score in the N-gram list is known, then no hashing is needed for the rest LM scores with this context, since they can be retrieved sequentially: if the stored word index matches the last word index of the current N-gram, then get the score; otherwise, the N-gram does not exist, and its score is approximated by a backing-off to N-1-gram.

Now LM context building becomes the problem of locating the first LM score for a given LM context in the N-gram list. A natural choice is to use MPHF to map a given LM context to the index of the first N-gram with this LM context[21] in the N-gram list. In this way, the number of keys of MPHF becomes only the number of different LM contexts in the N-gram list, which is much less than the total number of N-grams. In other words, in OPCP, MPHF is no longer needed for N-gram hashing, instead it is only needed for the simplified task of locating the first N-gram with a LM context, and as the result, less number of keys is needed. For example, the WSJ 20K task has as many as 3.5 million trigrams, but only 0.76 million LM contexts; SWB 33K task has nearly 10 million trigrams, but only 1.98 million LM contexts. In addition to cutting down the number of hashing operations from the vocabulary size (20K and 33K) to only one for each LM context, the reduced number of hashing keys would allow integer-based MPHF in both tasks for very

[20] This is true for most N-gram format, e.g. ARPA format. If N-grams are in other format, it is easy to re-sort them in alphabetical order.
[21] Binary search is another choice to locate the first N-gram of a given LM context, but the speed will be much slower than MPHF.

fast and memory-efficient hashing operations.

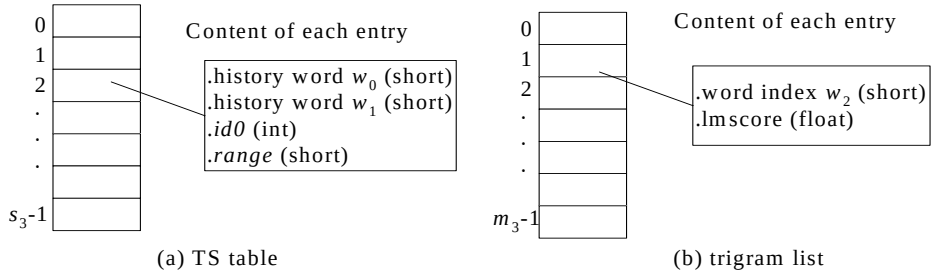

Figure 4.6. The TS table and trigram list used in OPCP method (s3 = the number of different word histories in trigram list, m3 = trigram size)

The data structures of the new MPH table and the trigram list used in OPCP is shown in Figure 4.6. The new MPH table is referred to as Trigram Start index table or in short, the TS table. For the TS table of Figure 4.6(a), each entry includes four elements, where w_0 and w_1 are the history words of a LM context, *id0* is the index of the first trigram with the word history (w_0, w_1) in the trigram list, and *range* is the total number of trigrams with this word history. In the trigram list of Figure 4.6(b), only the last word index (w_2) and the trigram score are needed for each entry. Compared with the trigram list in MPH-only or LMCP methods (see Figure 4.1), each trigram entry will save 4 bytes for vocabulary size less than 64K, or 8 bytes for vocabulary size larger than 64K.

Not every trigram lookup can successfully retrieve a trigram score, since the number of trained trigrams is far less than the cube of vocabulary size. Back-off is commonly used to handle the missing trigrams. For the back-off formula (4.7), both $P(w_2 \mid w_1)$ and $\alpha(w_0, w_1)$ require looking up bigrams, and so bigram access needs to be fast. Similar to the

LM context array used for trigrams, a bigram context array is used to store the bigram scores for each given bigram context. Although the method for trigrams can be similarly used here, i.e., use a MPH table to locate the first bigram with each given context in bigram list, a more efficient organization is possible. Since LM context for bigram has only one history word, an array with the size of vocabulary can be used to store the information. For each entry in this array, only the index of the first bigram in the bigram list needs to be stored. Since words are sequentially indexed by 0, 1, …, V-1, the value of *range* does not need to be stored, instead, it can be obtained by the difference of *id0* values of two neighboring contexts. This saves memory for bigram lookup. This array of bigram *id0* values is referred to as Bigram Start index table, or in short the BS table.

Figure 4.7 shows the data structure of BS table (a), and the bigram list (b). For each entry of the bigram list, only the last word index is needed as in the trigram list, while an additional float-type data is needed to store the back-off parameter.

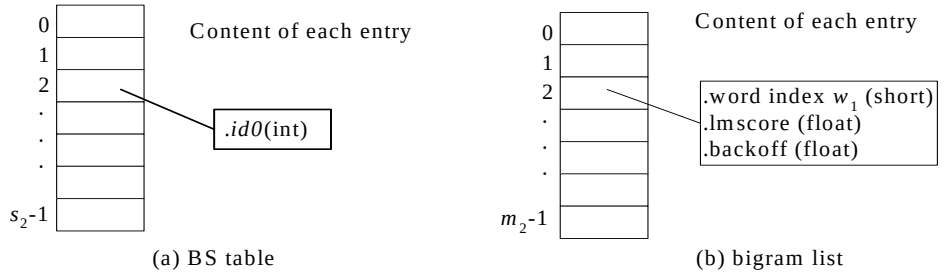

Figure 4.7. The BS table and bigram list used in OPCP method (s_2 = V = vocabulary size, m_2 = bigram size)

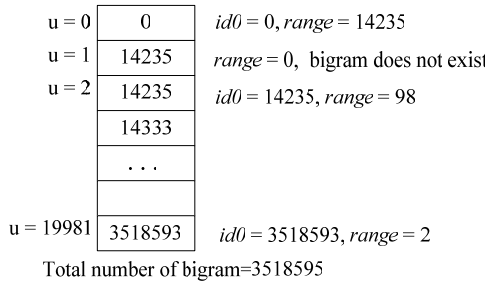

Figure 4.8. Obtaining range value from two neighboring entries of id0 values in BS table

Figure 4.8 illustrates the way the *range* value is computed from two neighboring values of *id0* in the BS table. There are two special cases. Firstly, if for any word history the bigram score does not exist in the bigram list, then the *id0* of this word history is set as the *id0* of the next word history (e.g., the second entry in the BS table in Figure 4.8), so that its *range* value is zero after taking difference of the current and the next *id0*s. Secondly, for the last entry of the array, the *range* value is obtained by subtracting total bigram number by current *id0*.

Figure 4.9 illustrates the procedure of building a trigram context for a given LM context (u, v). There are three steps. In Step 1, it checks if the trigram context array already has this context (u, v), if yes, exit; otherwise, enter the next Step. In Step 2, it looks up the TS table by hashing with the key (u, v), and obtains *id0* and *range*. In Step 3, it enters a loop to look-up trigram scores for (u, v, w), w = 0, ..., V-1, where V is the vocabulary size. Starting from *id0*, it sequentially gets trigram scores in the trigram list, until a total of *range* number of trigrams are retrieved. There are two possible cases in the loop, i.e., directly getting the trigram score or backing off to bigram. Backing-off occurs when the last

word *w* does not match the word in the current entry of the trigram list, or the number of obtained trigrams has already reached the value of *range*. In Figure 4.9, the procedure for backing-off to bigram is omitted due to its similarity to the trigram case, the difference is that the hashing operation in Step 2 is replaced by a simple array reference and a subtraction operation is used to get the value of *range*.

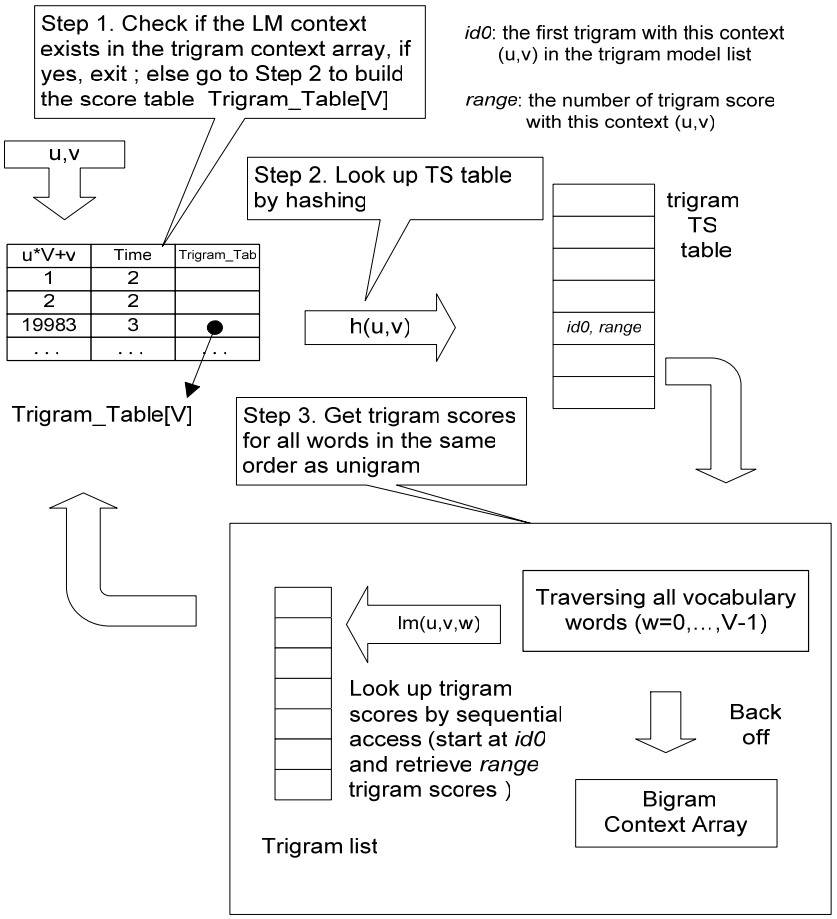

Figure 4.9. The procedure of building a LM context for word history (u, v) in OPCP

4.5 Experimental Results

In this section, three phases of experiments were conducted to demonstrate the advantages of OPCP described in previous sections. The three LMs used in the experiments are listed in Table 4.3, where m_i (i= 1, 2, 3) is the size of i-gram, and s_3 is the number of distinct LM contexts in trigram. Table 4.4 also gives the pruning thresholds for WSJ 20K and SWB 33K tasks (same pruning thresholds used for two SWB 33K tasks with large and small LMs).

The experimental platform is a Dell Xeon workstation with a single CPU of 3.2 GHz clock rate and 3GB memory space.

Table 4.3. Three LMs used in experiments, where m_i (i = 1, 2, 3) is the size of i-gram, s_3 is the number of different LM context in trigram.

Task	m_1	m_2	m_3	s_3	Size Category
WSJ 20K	19,982	3,518,595	3,153,527	767,765	Medium
SWB 33K	33,216	3,246,315	9,966,270	1,981,264	Large
SWB 33K-simplified	33,216	319,938	137,938	40,079	Small

Table 4.4. Pruning thresholds for WSJ 20K and SWB 33K tasks (same pruning thresholds used for SWB 33K tasks using large or small LM).

Task	b_s	b_w	N_s	N_w	b_{PLA}	b_{xwd}
WSJ 20K	240	100	18,000	15	245	150
SWB 33K	145	40	8,500	15	205	100

4.5.1 Performance Comparison of Different MPHFs for LM Lookup

In the initial phase of experiments, the SWB 33K task was evaluated with the simplified LM. The aim of the initial phase was to compare the time-memory performances of different MPHFs for LM lookup. As discussed in Section 4.2, when the number of keys is large, only string-key based MPHFs are available on 32-bit computer architecture. An empirical rule is that when the number of keys exceeds 2 million, integer-key based MPHFs would be difficult to train. To be able to compare integer-key based MPHF with string-key based MPHF, the simplified LM model was therefore used for the SWB task.

Table 4.5. Comparison of training time (Time) and memory cost (Space) for different MPHFs on SWB 33K task with a simplified trigram language model (memory cost does not include the storage of N-grams LM)

	Bigram		Trigram	
	Time (secs)	Space (Bytes)	Time (secs)	Space (Bytes)
#key		319 938		137 938
IMPHF	112.5	0.50 M	6.1	0.25 M
SMPHF	48.7	1.14 M	6.0	0.34 M
OP-SMPHF	171.2	2.53 M	397.2	1.19 M

Table 4.5 gives the training time and memory cost of bigram and trigram MPHFs for IMPHF, SMPHF, and OP-SMPHF, where memory cost does not include the storage of N-gram LM. As shown in the table, in both cases of bigrams and trigrams, SMPHF took the least training time, and it took less than half the storage space of OP-SMPHF. IMPHF took the least storage space, i.e., its total storage is 0.73 MB less than SMPHF and 2.97 MB

less than OP-SMPHF.

Tables 4.6 and 4.7 give the comparative results for LM lookup in a decoding experiment. The 776-sentence female testing set was used. The overall decoding time and LM lookup time are provided in Table 4.6, and the overall space and the LM lookup space are provided in Table 4.7. The results from binary search [Deshmukh+ 99] and from non-perfect hashing are also given as references. The threshold parameters of the decoder were kept the same for these methods. For binary search, the same implementation as in ISIP's one-pass decoder (version 5.15) [Deshmukh+ 99] was integrated into TigerEngine, and its detailed data structure was described in [Zhang&Zhao 02]). The non-perfect hashing method as used in HTK 3.0 was also integrated into TigerEngine. It is worth noting that in HTK 3.0, the non-perfect hashing was used in offline network building, with only bigram model supported. HTK 3.0 uses a token-passing algorithm based on a pre-compiled static recognition network [Young+ 89] instead of tree-copy based Viterbi time synchronous decoding. Therefore the non-perfect hashing result in Tables 4.6 does not reflect the real LM lookup time in HTK 3.0.

Table 4.6. Comparison of LM score retrieval time (xRT) for different MPHFs on SWB 33K task with the simplified LM

Method	Bigram LMLA (Acc=54.76%)		Trigram LMLA (Acc=54.96%)	
	LM Lookup	Overall	LM Lookup	Overall
IMPHF	1.060	2.107	1.669	2.743
SMPHF	2.040	3.132	3.908	5.040
OP-SMPHF	2.691	3.821	6.089	7.270
Binary Search (ISIP)	4.034	5.382	6.811	7.897
Non-perfect Hash (HTK)	4.309	5.497	10.410	11.631

Table 4.7. Memory Cost (MB) of LM Lookup for different methods in SWB with simplified LM (WER = 45.04% for trigram LMLA and 45.24% for bigram LMLA). Note that memory cost is the same for trigram LMLA and bigram LMLA

Method	Trigram/Bigram LMLA	
	LM Lookup	Overall
IMPHF	5.98	134.02
SMPHF	6.71	134.75
OP-SMPHF	8.95	137.89
Binary Search (ISIP)	7.52	137.14
Non-perfect Hash (HTK)	11.34	140.23

It can be seen in Table 4.6 that the three MPHFs are all much faster than binary search and non-perfect hashing for LM lookup. For trigram LMLA, IMPHF is faster than SMPHF and OP-SMPHF by factors of 2.3 and 3.7, respectively, and SMPHF is faster than OP-SMPHF by a factor of 1.6. Based on results of memory cost in Table 4.7, IMPHF also wins with the smallest memory. SMPHF is second to IMPHF. OP-SMPHF takes the largest memory among the three MPHFs, and its memory usage is also larger than binary search by 1.43 MB. In the subsequent experiments, IMPHF or SMPHF is used as the default method for MPH-based LM lookup.

Tables 4.6 and 4.7 also give results for LM lookup by bigram LMLA instead of trigram LMLA. Note that the memory cost of bigram LMLA was the same as trigram LMLA for MPH-only method, since trigram MPH was needed in word-ending LM lookup for both cases. Table 4.6 shows that on one hand, bigram LMLA was faster but less accurate than trigram LMLA, and on the other hand, the faster the LM lookup method, the smaller the time difference between bigram and trigram LMLAs. For example, in non-perfect

hashing (the last row of Table 4.6), bigram LMLA (4.309xRT) is faster than trigram LMLA by 6.101 xRT, but in IMPHF (the first row of Table 4.6), bigram LMLA (1.060 xRT) is faster than trigram LMLA by only 0.609 xRT. It is shown in Section 4.5.3 that when LM lookup speed is further improved by the proposed OPCP method, bigram LMLA becomes slower than trigram LMLA, which justifies using trigram LMLA instead of bigram LMLA for both fast and accurate decoding.

4.5.2 Comparison of MPH, LM Cache, LMCP, and OPCP for LM Lookup

Comparative experiments on the WSJ and SWB tasks were conducted to evaluate the discussed LM lookup methods, including MPH-only, node-based LM cache, LMCP, OPCP, as well as certain combinations of these methods. The threshold parameters of the decoder were kept the same for these methods in each task. The speed for LM lookup (xRT) and the memory cost for LM lookup and LM storage (in Mega Bytes or MB) were compared. The three LMs of Table 4.3, representing small, medium and large LMs, were used in the experiments.

4.5.2.1 WSJ 20K Task (Medium LM)

Table 4.8 provides the experimental results in processing speed for WSJ 20K task. Word error rate was fixed to 9.73% in this task with the same pruning thresholds for all the

methods investigated. The MPH-only method was firstly used (1st row), where the LM lookup took 1.180 xRT. Next, the node-based LM cache was combined with MPH method (2nd row), which reduced the LMLA time by about 57% and took 0.513 xRT in LM lookup. Here the cache size was set as 40 for each node, which was empirically optimized for both fast speed and small memory cost. Thirdly, LMCP method was combined with MPH (3rd row), which took 1.048 xRT for LM lookup. It is observed that although LMCP reduced about 89% of LMLA time from MPH+Cache, the overhead in LM context building was almost twice the LMLA time in MPH+Cache, and the net effect of LMCP therefore made LM lookup much slower than MPH+Cache, with only a minor improvement over the MPH-only method. Fourthly, LM cache was combined with the LMCP method (4th row), where it took 1.082 xRT for LM lookup, meaning that by combining LMCP with LM cache, LM lookup became slower. It is again due to the overhead time spent in LM context building. In both cases of MPH+LMCP and MPH+LMCP+Cache, the average number of new LM contexts per frame was 0.78, larger than what was reported for a different task, i.e., 0.25 contexts/frame, in [Cardenal+ 02].

The proposed technique of OPCP was then evaluated. In OPCP (5th row), LM lookup took only 0.080 xRT. Consistent to the above analysis, the LM context building time in OPCP was reduced to a very small number, i.e., 0.024 xRT, which was only 2.4% of context building time in LMCP method. LM lookup took 8.2% of the overall decoding time, much less than 53.4% in MPH-only, 34.6% in MPH+Cache and 52.3% in MPH+LMCP.

As the result, the decoder achieved real-time speed with trigram LMLA.

An interesting result was provided by OPCP+Cache in the last row of Table 4.8. It shows that LM cache did not further cut down LMLA time of OPCP, where its LM lookup took more time than OPCP by 0.016 xRT. A similar behavior is also observed when comparing MPH+LMCP with MPH+LMCP+Cache, where LM cache increased the time of LMLA. This will be analyzed in Section 4.5.3.4.

Table 4.8. Comparison of LM lookup time (xRT) in WSJ 20K task (WER = 9.73%)

Method	LM Lookup				Decoder Overall
	Context Building	LMLA	LM	LM Overall	
MPH-only	0	1.169	0.011	1.180	2.210
MPH+Cache	0	0.498	0.015	0.513	1.481
MPH+LMCP	0.989	0.057	0.002	1.048	2.005
MPH+LMCP+Cache	1.006	0.074	0.002	1.082	2.060
OPCP	0.024	0.054	0.002	0.080	0.980
OPCP+Cache	0.025	0.069	0.002	0.096	1.013

Table 4.9. Comparison of memory cost (MB) in WSJ 20K task (WER=9.73%)

Method	LM Lookup					Decoder Overall
	LM Storage	MPHF/ TS+BS	LM Context	LM Cache	LM Overall	
MPH-only	70.5	16.2	0	0	86.7	204.6
MPH+Cache	70.5	16.2	0	10.1	96.8	215.7
MPH+LMCP	70.5	16.2	7.9	0	94.6	211.4
MPH+LMCP+Cache	70.5	16.2	7.9	10.1	104.7	222.9
OPCP	51.8	9.4	10.2	0	71.4	196.7
OPCP+Cache	51.8	9.4	10.2	10.1	81.5	207.7

In Table 4.9, memory costs for both LM lookup and overall decoding system are compared. The number of nodes of the lexical tree for WSJ 20K task was 133K,

and the compressed lexical tree with 33K nodes was used for LM cache. The node-based cache took 10.1 MB of memory. For LMCP and OPCP methods, the maximal number of active trigram contexts in a time frame was 104, and the maximal number of bigram contexts was 15.

It should be noted that in Table 4.9, the memory cost of each decoder component represents only an estimate, which is proportional to but is not the exact memory size taken by an online system where memory allocation is dependent on specific mechanism of memory management, usually by pages. The overall memory cost of each decoding task was recorded from the Task Manager of Windows system on peak memory usage, which represents the actual total memory cost. This rule also applies to memory cost measurements in other tables.

It can be seen from Table 4.9 that the proposed OPCP method used the least memory in LM lookup (71.4 MB), even less than the MPH-only method (86.7 MB). LM cache (96.8 MB) and LMCP (94.6 MB) both used more memory than the MPH-only method. Therefore, OPCP achieved fastest speed and smallest memory usage. The first reason of OPCP's memory saving is in LM storage, because only the last word index is needed for each N-gram in both trigram and bigram lists. The second reason is that the memory usage by MPHF is also reduced because the number of keys is much smaller than the rest methods and only trigram TS table needs MPHF, and bigram BS table no longer needs MPHF. Although the LM context arrays for trigrams and bigrams took an extra 10.2 MB

memory than MPH-only method, the increment was less than the 25.5 MB reduction in LM storage and MPHF tables (including TS and BS tables), and so the net effect is a saving of 15.3 MB memory in the decoder in OPCP than MPH-only method. Note that the memory usage for LM lookup in OPCP (71.4 MB) was only slightly higher than the original LM storage space (70.5 MB). The total memory for OPCP was 196.7 MB, which was the least among all the studied methods.

4.5.2.2 SWB 33K Task (Large LM)

Tables 4.10 and 4.11 show the results obtained on SWB 33K task with the large LM. Word error rate was fixed to 43.81% in this task with the same pruning thresholds for all the methods investigated. Some differences from the results of WSJ 20K task in Table 4.8 and Table 4.9 are worth noticing. In Table 4.10, the MPH-only method (1st row) took 4.701 xRT in LM lookup, which was 80.6% of the total decoding time. In WSJ 20K task, however, only 53.4% of decoding time was spent in this part, indicating that the increase of vocabulary size significantly increased LM lookup time. With the task vocabulary going from 20K to 33K, the number of lexical nodes grew from 133K in WSJ 20K task to 201K in SWB 33K task, and the number of nodes in the compressed lexical trees also increased from 33K to 52K. As the result, the average number of LMLA lookups per frame increased significantly. However, in MPH+Cache method (2nd row), the LM lookup time was cut down by a factor of about 7.7 to 0.608 xRT, due to the speedup in LMLA by LM cache. The

number of entries in each node-based cache was empirically set as 45 for both fast access speed and small memory cost. The LMCP method (3rd row) also cut down the LM lookup time significantly, by a factor of 5.2. In the method of MPH+LMCP+Cache (4th row), LM lookup was slightly faster than MPH+LMCP, but still much slower than MPH+Cache, again due to the heavy overhead of LM context building. The average number of new LM context per time frame was 0.357, which was also larger than the 0.250 new context/frame number reported in [Cardenal+ 02].

In our proposed OPCP method (5th row), LM lookup time was cut down to only 0.150xRT, with a speedup factor of 4.1 compared with LM cache method, and a speedup factor of 31.3 compared with MPH-only method. Finally, there was no further advantage when OPCP was combined with LM cache (6th row), compared with OPCP alone. For the SWB 33K task with large LM, OPCP reduced the LM lookup time from 80.6% of total decoding time to 14.1%, where the smaller speedup factor of LM lookup as compared with what was obtained in WSJ 20K is due to the increase of vocabulary size that increased computations in LM context building as well as the amount of maximization operations in LMLA. Now the overall decoding time using OPCP approached real-time (1.063 xRT), compared with 5.835 xRT, 1.586 xRT, and 1.880 xRT in MPH-only, MPH+Cache, and MPH+LMCP methods, respectively.

Table 4.10. Comparison of LM lookup time (xRT) in SWB 33K task (Large LM, WER = 43.81%)

Method	LM Lookup				Decoder Overall
	Context Building	LMLA	LM	LM Overall	
MPH-only	0	4.684	0.017	4.701	5.835
MPH+Cache	0	0.593	0.016	0.608	1.586
MPH+LMCP	0.776	0.133	0.003	0.912	1.880
MPH+LMCP+Cache	0.763	0.134	0.003	0.901	1.877
OPCP	0.022	0.125	0.003	0.150	1.063
OPCP+Cache	0.022	0.128	0.003	0.153	1.070

Table 4.11. Comparison of memory cost (MB) in SWB 33K task (Large LM, WER=43.81%)

Method	LM Lookup					Decoder Overall
	LM Storage	MPHF/ TS+BS	LM Context	LM Cache	LM Overall	
MPH-only	132.5	40.2	0	0	172.7	319.8
MPH+Cache	132.5	40.2	0	18.0	190.7	340.7
MPH+LMCP	132.5	40.2	7.5	0	180.2	330.3
MPH+LMCP+Cache	132.5	40.2	7.5	18.0	198.2	346.7
OPCP	88.2	22.9	11.0	0	122.1	271.1
OPCP+Cache	88.2	22.9	11.0	18.0	140.1	299.6

In Table 4.11, memory costs are shown for different methods similar to those in Table 4.9. Since the cache size was 45 and the number of nodes in the compressed lexical tree was 52K, the memory cost for LM cache was 18.0 MB. The most notable result in Table 4.11 is that in OPCP, the total memory cost for LM lookup and LM storage (122.1 MB) became less than the memory cost of original LM storage (132.5 MB), with a saving of 10.4 MB. The maximal number of active trigram and bigram contexts in a time frame was 59 and 14, respectively. Because the numbers of trigrams and bigrams used in this task

were much larger than that in WSJ 20K, the effectiveness of OPCP in memory saving was more pronounced. It should be noted that, for every bigram entry, OPCP saved 2 bytes over other methods, and for every trigram entry OPCP saved 4 bytes over other methods, given that the task vocabulary size was less than 64K and short integer was used for word indices. If the vocabulary size is larger than 64K, then more memory saving is expected by the OPCP method.

4.5.2.3 SWB 33K Task (Small LM)

Tables 4.12 and 4.13 show the results on SWB 33K task with the small LM. Word accuracy was fixed to 54.96% in this task, using the same pruning thresholds as the large LM task. Because of the small number of keys, IMPHF was used for MPH-based methods in contrast to the SMPHF used in the WSJ task and SWB task with large LM. In Table 4.12, it can be seen that the method of IMPHF-only (2st row) was faster than SMPHF-only method (1st row) by a factor of 2.3, and it took 1.669 xRT for LM lookup which was 60.8% of total decoding time. The IMPHF+Cache method (3nd row) cut down LM lookup time to 0.291 xRT, with a speedup factor of 5.7 over IMPHF-only. The size of LM cache was empirically optimized to be 30, which was smaller than the size of 40 in WSJ 20K task (medium LM) and 45 in SWB 33K (large LM). IMPHF+LMCP method (4th row) also cut down LM lookup time into 0.370 xRT with a speedup factor of 4.5 over IMPHF-only method. By combining LMCP with LM cache (5th row), the LM lookup time became 0.360 xRT, which

was slightly faster than IMPHF+LMCP, but still slower than IMPHF+Cache. Here the average number of new LM context per frame was 0.357, same as SWB 33K with large LM. Our proposed method, OPCP(6th row), further cut down the LM lookup time to 0.144 xRT, with speedup factors of 11.6 over IMPHF-only, 2.0 over IMPHF+Cache, and 2.6 over IMPHF+LMCP. Finally, in OPCP+Cache, the LM lookup time was 0.140 xRT, which was slightly faster than OPCP, but the margin (0.004 xRT) was insignificant. This is different from the results in previous two tasks, where OPCP+Cache were slower than OPCP. The main reason is that the computation load of trigram lookup was much less for the small LM than the other two LMs, and therefore LM cache can further speedup LMLA on top of the OPCP method. The overall decoding time by using OPCP and OPCP+Cache both approached real-time (1.061 xRT and 1.059 xRT).

Table 4.12. Comparison of LM lookup time (xRT) in SWB 33K task (small LM, WER = 45.04%)

Method	LM Lookup				Decoder Overall
	Context Building	LMLA	LM	LM Overall	
SMPHF-only	0	3.891	0.017	3.908	5.040
IMPHF-only	0	1.661	0.009	1.669	2.743
IMPHF+Cache	0	0.282	0.009	0.291	1.233
IMPHF+LMCP	0.239	0.128	0.003	0.370	1.309
IMPHF+LMCP+Cache	0.238	0.120	0.003	0.360	1.302
OPCP	0.017	0.124	0.003	0.144	1.061
OPCP+Cache	0.017	0.120	0.003	0.140	1.059

Table 4.13. Comparison of memory cost (MB) in SWB 33K task (Small LM, WER=45.04%)

Method	LM Lookup					Decoder Overall
	LM Storage	MPHF/ TS+BS	LM Context	LM Cache	LM Overall	
SMPHF-only	5.23	1.48	0	0	6.71	134.8
IMPHF-only	5.23	0.75	0	0	5.98	134.0
IMPHF+Cache	5.23	0.75	0	12.0	17.98	148.0
IMPHF+LMCP	5.23	0.75	7.35	0	13.33	141.5
IMPHF+LMCP+Cache	5.23	0.75	7.35	12.0	25.33	155.5
OPCP	4.09	0.57	10.6	0	15.26	143.9
OPCP+Cache	4.09	0.57	10.6	12.0	27.26	157.7

In Table 4.13, memory costs by using different methods are shown. Since the cache size was 30 and the number of nodes in compressed lexical tree was 52K, the memory cost for LM cache was 12 MB. For OPCP, the memory cost of LM lookup (15.26 MB) became much larger than IMPHF-only method (5.98 MB) and the original LM storage (5.23 MB). The main reason is that the LM size in this task was much smaller than other two tasks. From Table 4.3, it is seen that the total number of N-grams of the small LM was only 7.34% of the medium LM used in WSJ 20K task and 3.71% of the large LM used in SWB 33K. In comparison with IMPHF-only, although OPCP still saved memory spaces in LM storage (4.09 MB vs. 5.23 MB) and in MPHF/TS+BS tables (0.57MB vs. 0.75 MB), the total saving in these two aspects (1.32 MB) was much smaller than the increased memory cost due to LM context building, which was 10.6 MB. Here the maximal number of trigram contexts per time frame was 58, and that of bigram was 13, both accounted for the memory cost of LM context. It is worth noting that the memory cost of OPCP+Cache was

12MB more than OPCP method. Given the comparable LM lookup time of OPCP+Cache and OPCP, it can be concluded that the OPCP method is still overally superior to the OPCP+Cache method.

4.5.3 Analyses and Discussions

In this section, analyses and discussions on the above experimental results are given.

4.5.3.1 Summative Comparison of LM Lookup Methods

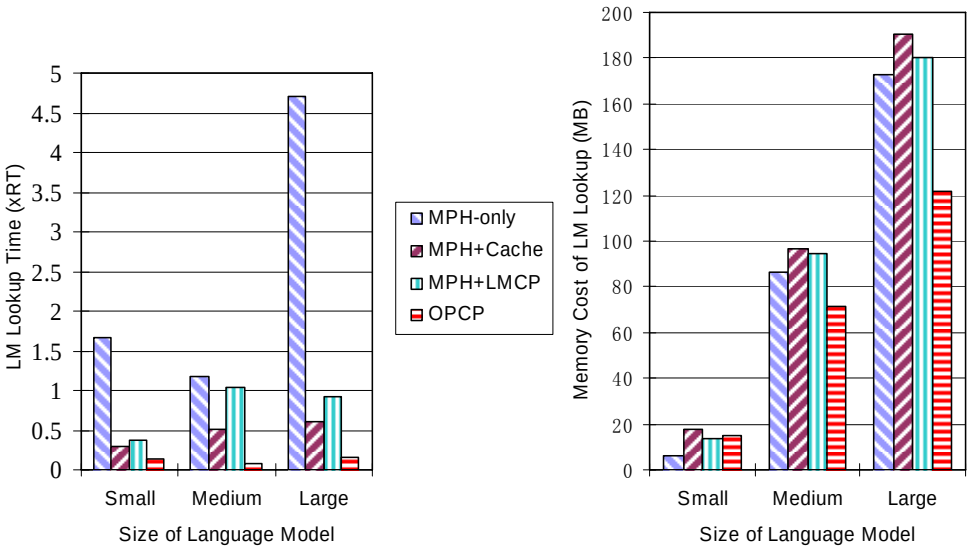

Figure 4.10. Comparison of LM lookup time and memory cost using four methods with three different sizes of LM

Figure 4.10 gives a summative comparison on performances of the four major LM lookup methods discussed in this paper, including MPH-only, MPH+Cache, MPH+LMCP, and OPCP. The left bar chart shows the LM lookup time (xRT), and the right bar chart shows the memory cost (MB) for LM lookup. From the left chart, it is observed that our proposed

OPCP method is very time efficient in comparison with the other three methods. From the right chart, it is observed that OPCP saved large memory space over the other three methods with medium and large sized LMs, but it took more memory space than MPH-only method with small sized LM. Note that in this case, the total memory cost of LM lookup using OPCP was less than 20MB, and LM lookup speed of OPCP was 11.5 times faster than MPH-only method. Therefore, OPCP is still superior over MPH-only method. It is also clear from the right chart that with the increase of LM size, the memory usage for LM lookup by all methods consistently increased, and the memory saving by OPCP also consistently increased.

4.5.3.2 Comparison of Decoding Performance using bigram LMLA and trigram LMLA

As mentioned in Section 4.5.1, when replacing trigram LMLA by bigram LMLA, LM lookup and overall decoding became faster, while word error rate became worse. However, when LM lookup is sufficiently fast, trigram LMLA could be faster than bigram LMLA. The comparative experiments using OPCP verified this hypothesis. Table 4.14 lists the total LM lookup time (RT), a breakdown of LM lookup time (RT), overall decoding time (RT) and word error rate (%) by using bigram LMLA and trigram LMLA for the three LMs when OPCP was used. As can be seen in Table 4.14, context building for bigram LMLA was faster than that for trigram LMLA (e.g., 0.009 xRT vs. 0.022 xRT in SWB 33K task with large LM) since only bigram context was needed. LMLA time was nearly the same for bigram and trigram LMLAs, due to the same maximization operations for bigram or trigram LMLA. However, at word-ending nodes (leave nodes), LM lookup in bigram LMLA needed MPH hashing instead of direct array reference as in trigram LMLA, and this part of bigram LMLA was slower than that of trigram LMLA (e.g., 0.020 xRT vs. 0.003 xRT in SWB 33K task with large LM). Although the amounts of overall LM lookup

time were about the same for the two methods, the overall decoding time by using bigram LMLA was slower than using trigram LMLA (e.g., 1.089 xRT vs. 1.063 xRT in SWB 33K task with large LM). The result is reasonable since by using trigram LMLA the linguistic knowledge used in the search was more precise and the best path hypothesis was more pronounced compared with competing hypotheses, which speeded up the search. By using OPCP, the trigram LMLA was better than bigram LMLA in both speed and accuracy for LVCSR.

Table 4.14. Comparison of decoding time (xRT) and WER (%) using Bigram LMLA and trigram LMLA (OPCP)

Task	LMLA method	LM Lookup (xRT)				Overall Decoding (xRT)	WER (%)
		Context Building	LMLA	LM	LM Overall		
WSJ 20K	Bigram	0.017	0.053	0.015	0.085	1.047	9.76
(Medium LM)	Trigram	0.024	0.054	0.002	0.080	0.980	9.73
SWB 33K	Bigram	0.009	0.125	0.020	0.154	1.089	44.10
(Large LM)	Trigram	0.022	0.125	0.003	0.150	1.063	43.81
SWB 33K	Bigram	0.007	0.126	0.020	0.153	1.085	45.24
(Small LM)	Trigram	0.017	0.124	0.003	0.144	1.061	45.04

4.5.3.3 Effect of Pruning Thresholds

In above experiments, results on decoding speeds (xRT) and word error rates (%) were obtained with fixed pruning thresholds for each task. It is interesting to see how the changes of pruning thresholds would affect performance of a decoding engine with our proposed methods for fast LM lookup.

Figure 4.11 illustrates the results of decoding performance (WER versus Speed) of the four major LM lookup methods under different working points associated with different pruning thresholds. From left to right the pruning thresholds changes from tight to loose. Here only the task of WSJ 20K is reported, while the same trend was observed in other two tasks.

The following trends are shown in Figure 4.11. Firstly, there is a trade-off between search speed and word error rate which is true for each method. For example, in OPCP method, when the pruning thresholds change from loose to tight, decoding time decreases from 2.77 xRT to 0.29 xRT, but the word error also increases from 9.37% to 12.88%. Secondly, at different working points, the proposed OPCP method is always the best, the MPH+Cache method is second to OPCP, then it comes MPH+LMCP method, and the slowest one is MPH-only method. This is consistent to our previous experimental results with fixed pruning thresholds.

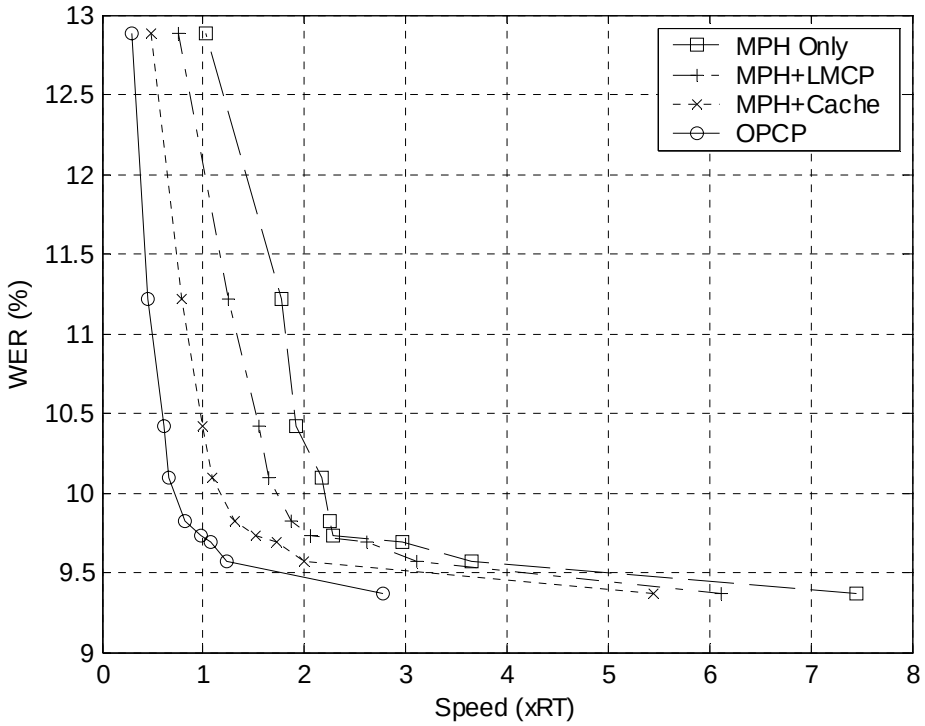

Figure 4.11. Comparison of performance for four different LM lookup methods under different working points (from left to right pruning thresholds changed from tight to loose).

4.5.3.4 Effect of LM Cache

A phenomenon observed in previous experiments is that when node-based LM cache was

combined with LMCP or OPCP method, it did not further speed up search, and instead in some cases (such as in WSJ 20K and SWB 33K-large LM) it slightly slowed down LM lookup. This is interesting because LM cache has been shown very helpful in speeding up LM lookup when combined with MPH-only method. In this section, we analyze the operations of LM cache used in our experiments and compare it with LM context array based methods such as OPCP.

During the time-synchronous search, whenever a LM lookup is demanded by a new LM context (u', v') at a node S, the decoder firstly checks the LM cache of node S to see if the context (u', v') already exists, which is simply done by checking the LM context value (u*V+v) in each cache item (see Figure 4.2). If the LM context value (u'*V+v') already exists in one cache item, i.e., a cache hit, the LM score is directly accessed. Otherwise a cache miss occurs and higher level operations of LM lookup are needed to get the LMLA score for the context (u', v') at the node S. With a cache miss, a replacement policy is needed to decide which cache item will be replaced by the new context. In MPH+Cache method, the decoder needs to initiate a series of hashing operations and a maximization operation to get the LMLA score. In OPCP+Cache method, the decoder will firstly check if the LM context already exists in LM context array. If so, then only a maximization operation is needed; otherwise, a new LM context will be built up by computing those LM probabilities for all vocabulary words based on the fixed LM histories (u', v'), and then a maximization operation is performed to get the LMLA score for node S. In tree-copy based search organization, a new LM context array is built whenever a new tree-copy appeared in search. Therefore, when a cache miss happens, only a maximization operation is needed to get the LMLA score for the current context (u', v') at node S, because the LM scores for context (u', v') are already stored in trigram context array (see Figure 4.2 and Figure 4.3).

Based on above analyses, it can be seen that whether or not LM cache can further speed up the LM lookup in OPCP method depends on the efficiency of cache search and replacement relative to LM score maximization. Although LM cache reduces

maximization operations in LM context array in cache hit, it also brings in overhead in cache miss, which includes cache search, cache replacement selection and some other operations to maintain the cache structure (initialization etc.).

Table 4.15. Comparison of LM Lookahead between OPCP and OPCP+LM Cache methods, with different cache search algorithms and replacement policies. (WSJ 20K task, WER = 9.73%).

Average number per frame		**OPCP**	**OPCP+LM Cache (Cache Size = 40)**					
			Linear Search			**Binary Search**		
			Random	**LRU**	**LFU**	**Random**	**LRU**	**LFU**
#Cache Access/frame		n/a	1279.3					
Cache Hit rate/frame		n/a	72.54%	73.08%	60.16%	72.54%	73.08%	61.23%
LMLA Initialize/frame (ms)		0.152	0.152	0.157	0.156	0.161	0.156	0.154
Cache Overhead /frame	**Initialize (ms)**	0	0.222	0.222	0.220	0.227	0.214	0.216
	Search (ms)	0	0.313	0.315	0.354	0.347	0.337	0.360
	Selection (ms)	0	0.028	0.039	0.078	0.049	0.064	0.103
	Sub-total (ms)	0	0.563	0.576	0.652	0.623	0.615	0.679
Maximization/frame (ms)		0.509	0.128	0.120	0.173	0.121	0.115	0.167
Total LMLA lookup/frame(ms)		0.661	0.844	0.853	0.980	0.905	0.886	1.001

Table 4.16. Comparison of the effects of LM Cache size (WSJ 20K task, WER = 9.73%).

Average number per frame		**OPCP**	**Cache Size in OPCP+Cache (Linear Serach+Random Replacement)**			
			40	80	120	160
#Cache Access/frame		n/a	1279.3			
Cache Hit rate/frame		n/a	72.54%	72.98%	73.06%	73.09%
LMLA Initialize/frame (ms)		0.152	0.152	0.156	0.157	0.152
Cache Overhead/frame	**Initialize (ms)**	0	0.222	0.225	0.221	0.224
	Search (ms)	0	0.313	0.364	0.402	0.424
	Selection (ms)	0	0.028	0.027	0.025	0.028
	Sub-total (ms)	0	0.563	0.616	0.648	0.676
Maximization/frame (ms)		0.509	0.128	0.119	0.121	0.112
Total LMLA lookup/frame(ms)		0.661	0.844	0.891	0.926	0.939

Table 4.15 compares the average time spent in LM lookahead per frame (in millisecond) on WSJ 20K task by using OPCP and OPCP+Cache methods, and it also compares the effect of LM cache with different cache search algorithms and different replacement policies. The word error rate was fixed as 9.73% for all methods. In Table 4.15, LM lookahead operation is decomposed into three steps, consisting of "LMLA Initialization," "Cache Overhead," and "Maximization," where "Cache Overhead" is further decomposed into three sub-steps, i.e., "Cache Initialization," "Cache Search," and "Cache Replacement Selection." Two search algorithms were considered: linear search and binary search. Three replacement policies were investigated including Random, Least-Recently-Used (LRU), and Least-Frequently-Used (LFU) [Bryant&O'Hallaron 03]. The average number of cache access and average cache hit rate per frame in cache-based methods are also given in Table 4.15.

Firstly, it can be concluded from Table 4.15 that, in WSJ 20K task, the overhead of LM cache made the LMLA of the OPCP+Cache method slower than the OPCP method. Although using LM cache reduced the time of maximization operations by more than 67%, the total time of LMLA increased by 28~51% because of the cache overhead.

Secondly, it can be seen from Table 4.15 that binary search made cache search and cache selection slower than linear search. The problem was not caused by the binary algorithm itself, but it was caused by the overhead in binary search, where sorting is needed before binary search and in linear search the sorting is not needed. In previously reported experiments, linear search were used.

Thirdly, it can be seen from Table 4.15 that different replacement policies have different effects on cache efficiency. Among the three different policies, LFU is the worst, Random and LRU are similar. Combined with linear search, Random selection is faster than LRU, but combined with binary search, LRU is faster than Random selection. In

previously reported experiments, random replacement policy were used.

Table 4.16 shows the effect of LM cache size (40, 80, 120, 160) on the same task with WER = 9.73%. As can be seen, cache hit rate slightly increased with the cache size, but the cache overhead also increased, and the net effect of increasing cache size is to make the LMLA lookup slower. Our experiments at other working points also had similar results.

4.5.3.5 Number of LM Context

The number of maximal LM contexts in a time frame affects memory consumption of decoding engine with OPCP method or LMCP method. In this work, the number of LM contexts (including trigram and bigram) was affected by pruning methods but was not controlled otherwise. Figure 4.12 illustrates the changes of maximal number of LM contexts in a frame (Trigram and Bigram), as well as the changes of speed and WER with different pruning thresholds using OPCP method in WSJ 20K task. From left to right the pruning thresholds changes from loose to tight (same as those working points in Figure 4.11, but in reverse direction of changes). As can be seen in the figure, by moving the working points from 0 to 2, the number of trigram contexts was largely cut from about 230 to 105, without apparent increase of WER, while the decoding speed increased from 2.77 xRT to 1.24 xRT. Next, in moving the working points from 3 to 5, the number of trigram contexts slightly decreased, still without apparent increase of WER, while decoding speed changed from 1.08 xRT to 0.82 xRT. When the number of LM contexts was reduced further by more aggressive pruning thresholds, WER increased significantly, but the number of LM contexts was not reduced as significantly. It is also worth noting that although all those pruning methods as mentioned in Section 4.5.1 can change the number of LM contexts in a frame, the word-ending pruning has the most significant effect.

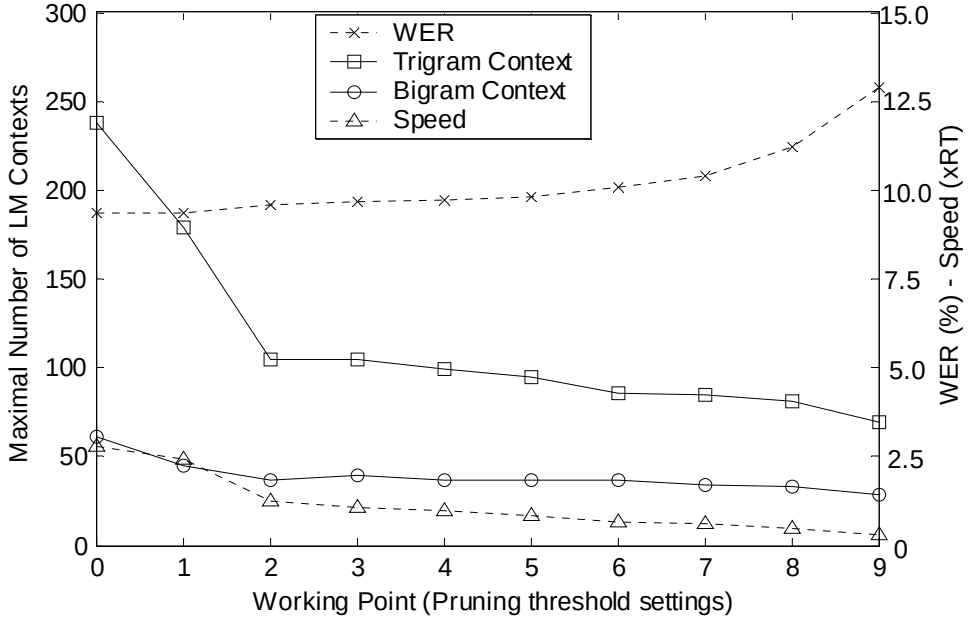

Figure 4.12. Maximal number of LM contexts (Trigram and Bigram) in a frame, decoding speed and WER versus working points defined by pruning thresholds in WSJ 20K task using OPCP (from left to right pruning thresholds changed from loose to tight).

4.5.3.6 Breakdown of Decoding Time

The distribution of decoding time on different components of the decoder when using OPCP and with fixed pruning thresholds as in Section 4.5.3 is given in Table 4.17, which provides insight to the time spent in different parts of decoder beside of LM lookup. Both times in xRT and in percentage are given for each task and each LM size. As can be seen, acoustic likelihood computing (AM) took about 33~36% of total decoding time, phone look-ahead (PLA) took 1.5% of total decoding time, and search management took about 51~54% of total decoding time.

Table 4.17. Breakdown of decoding time and word accuracy when using OPCP (AM = acoustic likelihood computing, LM = LM lookup, PLA = phone look-ahead, Search = search management)

Task	Time Breakdown(xRT/%)				Overall	Acc
	AM	LM	PLA	Search	Time (xRT)	(%)
WSJ 20K	0.355	0.080	0.016	0.528	0.980	90.27
(Medium LM)	36.27%	8.20%	1.64%	53.90%		
SWB 33K	0.359	0.150	0.015	0.539	1.063	56.19
(Large LM)	33.73%	14.15%	1.46%	50.66%		
SWB 33K	0.356	0.145	0.016	0.544	1.061	54.96
(Small LM)	33.52%	13.71%	1.51%	51.27%		

4.5.3.7 Effect of Search Algorithm

As pointed out previously, the above experimental results were obtained under tree-copy based search organization. An alternative search algorithm is single-tree based organization [Alleva 97, Soltau+ 02], where only one instance of lexical tree is used and different linguist histories are attached to each lexical arc. Since LM lookup is separated from search management where data structures of compressed LM tree and full lexical tree are separately maintained in decoding engine, fast LM lookup method such as OPCP can be used in different search algorithms including single-tree or tree-copy based algorithms.

The main motivation of single-tree based search algorithm is to reduce search effort by removing redundant hypotheses such as in Subtree Dominance [Allev+ 96, Ortmanns+ 98]. A logical assumption might be that the single-tree based algorithm can largely reduce the effort on LM lookup in tree-copy based algorithm, such that the benefits of the proposed OPCP method would no longer be as significant as reported above. However, the redundant hypotheses in Subtree Dominance can also be removed in tree-copy based algorithm, either implicitly by aggressive pruning and phone lookahead [Sixtus 03], or explicitly by the min-max algorithm [Ortmanns+ 98]. Our experiments indicated that even when Subtree Dominance was explicitly exploited as did in [Ortmanns+ 98], decoding speed was not further improved because most of those redundant paths had already been pruned out by regular pruning methods and phone lookahead. Studies in [Sixtus 03]also

supported this observation. On the other hand, maintaining different linguistic histories in each arc of a single tree incurs computational overheads that are not needed in tree-copy based search, such as insertion, deletion and maximization operations for heap-based single-tree algorithm [Alleva 97]. Therefore, although single-tree based search algorithm may reduce efforts in LM lookup, its benefit to decoding speed is partially compromised by the increase of search management.

4.6 Chapter Conclusion

In this chapter, a novel LM lookup method called Order-Preserving LM Context Pre-computing (OPCP) is presented to speed up N-gram LM lookup in large vocabulary continuous speech recognition. The idea came from the Minimum Perfect Hashing (MPH) based LM lookup methods proposed in recent years. Although the MPH methods can reduce LM lookup time by large factors as compared with conventional binary search and non-perfect hashing methods, LM lookup still takes significant time in LVCSR decoding when N-gram look-ahead such as trigram LMLA is used to avoid losing word accuracy. Several other methods were proposed recently to further reduce LM lookup time, including node-based LM cache and LM Context Pre-computing (LMCP). The node-based LM cache method can cut down the time of LM look-ahead by factors of about 2~7, with an increased memory cost. For the LMCP method, although it was very efficient in reducing LM look-ahead time, its large overhead in LM context building actually counteracts its time saving in LMLA, and it requires more memory space than LM cache method. Based

on studies of those methods, OPCP is proposed in the current work to achieve better performance in both speed and memory for LM lookup. In the aspect of speed, OPCP cuts down the hashing operations in LMCP from the size of vocabulary to only one for each LM context, and with the reduction of number of keys in MPHF, integer-key MPHF is obtainable which is faster than string-key based MPHF, in general. In the aspect of memory space, OPCP saves LM storage space by storing only the last word index and by using trigram MPHF with smaller number of keys to locate the first trigram of a given LM context in N-gram list. Experimental results showed that the proposed OPCP method is very effective in speed across different decoding tasks, from WSJ 20K to SWB 33K, and across different sizes of N-gram models, from hundreds of thousands to ten million trigrams. OPCP is also proven very efficient in memory usage for medium and large sized LMs, but it needs more memory than MPH-only and MPH+LMCP methods for small sized LM. The time and memory efficiencies of the OPCP method become more significant with the increase of LM size.

Chapter 5

Conclusions and Future Work

In this dissertation, a fast decoding engine, TigerEngine, was developed for speaker independent large vocabulary continuous speech recognition (LVCSR). The system is based on one of the most successful search algorithms for LVCSR, i.e., Viterbi time-synchronous beam search algorithm, which is in turn based on Dynamic Programming and Viterbi approximation using the best HMM state sequences to approximate the best word sequence. One advantage of Viterbi search is that its time-synchrony enables easy comparing and pruning different hypotheses at each time interval so that several heuristic pruning techniques including beam pruning and histogram pruning at both state- and word-level can be used to make the search space as compact as possible. Lexical tree based search structure is used in TigerEngine, by which a large factor of search space is reduced by merging the same prefix for all of words in vocabulary. During the search, those word sequence hypotheses from different word histories are organized in tree-copies. List-based data structure is used to assure direct access and better management of search hypotheses at different levels. To reach both the high

accuracy and fast speed, following specific methods are used in TigerEngine:

Crossword triphone based acoustic model.

Crossword triphone can better model the coarticulation phenomena between two neighboring words in continuous speech, that is, the last phoneme of a word may have different acoustic property with different following words. In contrast, within-word triphone does not consider the influence of next word to the last phoneme of current word and all word-ending triphone are actually modeled by a biphone model without right-context. By using crossword triphone in TigerEngine 1.0, word accuracy is significantly increased. The experiments in Section 3.2 for WSJ 20K task shows that up to 12.3% error reduction rate can be obtained by crossword model compared with within-word model.

Optimization for cross-word triphone based search structure.

With the use of cross-word triphone, the search space significantly increases which is mainly represented by a large number of fan-out arcs at word boundaries. To limit the search space, three effective methods are implemented including recombination after first-layer, cross-word language model lookahead, and optimization in lexical prefix tree.

Fast language model lookup with trigram language model.

Different from many reported systems of LVCSR, language model knowledge is integrated into search paths of TigerEngine in the form of trigram language model lookahead, instead of the approximation by bigram language model, which results in

higher accuracy (experimental results were given in Section 4.5.2.4). To reduce the computation of trigram language model lookup, a novel language model lookup method called Order-Preserving Language Model Pre-computing (OPCP) is proposed and experimented on two benchmark tasks, i.e., Wall Street Journal 20K and Switchboard 33K. Significant speedup is observed for language model lookup in each task without any loss of word accuracy. Another wonderful property of this new algorithm is that it saves much memory space especially for large language model with several millions or tens of millions of trigrams, which are typical sizes for LVCSR applications.

With the above methods, TigerEngine 1.0 achieves real-time decoding for both WSJ 20K and SWB 33K tasks, with little loss of accuracy compared with the best achievable word accuracy rate. The word accuracy for WSJ 20K also reaches the state-of-the-art level.

Potential future extensions to this dissertation work are listed as follows:

Integer-key based MPHF for large size of N-grams. As mentioned in Chapter 4, the integer-key based MPHF is significantly advantageous in both memory space and computation over string-key based MPHF. Although by using OPCP method the number of key of MPHF is already reduced by 75%~80%, it still may exceed 2 million in the future which is the empirical upper-bound for integer-key based MPHF on current 32-bit computer architecture. On the other hand it is possible to use 64-bit integer in IA-64

architecture or SSE2 based instructions in IA-32 system, and so the larger size of integer-key based MPHF may be obtained. The latter approach (SSE2) is very promising by using current experimental platform[22].

Further speed up TigerEngine by using single-tree based search organization. Single tree based search [Alleva+ 96, Slotau+ 02] is a very efficient method for search organization. Different from tree-copy based structure, it only uses one lexical tree to manage hypotheses from different word histories, and some smart algorithms are used to efficiently organize those hypotheses. One advantage of single-tree based search organization is that it may avoid redundant arcs in the search space, which exists in tree-copy based search structure. Subtree-Dominance [Ortmanns+ 98] is one of such examples. Initial implementation of single-tree based structure in TigerEngine for within-word model has already demonstrated significant advantage in speed than currently used tree-copy based structure with no loss of accuracy. But for crossword model based search, the initial implementation of single-tree structure seems much slower than tree-copy based structure. Future efforts can be made to improve the implementation and to make single-tree based organization work for crossword based search.

I**ncrease the vocabulary size beyond 64K.** Currently, all reported experiments of TigerEngine 1.0 are based on the vocabulary size smaller than 64K. It is very important to extend the vocabulary size to larger than 64K because that is the size of many real-world

[22] As mentioned before, it is already possible to use 64-bit integer type (__int64) with Visual C++ 6.0.

applications such as broadcast news captioning and spontaneous speech recognition[23].

Work toward spontaneous telehealth conversational speech. The ultimate goal of TigerEngine is to be used for real-time captioning of spontaneous telehealth conversational speech. Spontaneous speech is much difficult in recognition than read speech or broadcast news speech, due to its casual speaking style which leads to abundant phone lost or a lot of incomplete pronunciations as well as speaking rate changes and topic transfers. Experimental results in Chapter 4 on Switchboard task has already shown its difficulty compared with read speech of WSJ task, indicated by much lower word recognition rate (56% vs. 90%). To reach higher accuracy for spontaneous speech recognition, many complex acoustic processing and modeling techniques are necessary such as Vocal Tract Length Normalization (VTLN) [Wakita 77], Discriminative Training [Chou+ 96, Woodland&Povey 02], Heterogeneous Linear Discriminant Analysis (HLDA) [Kumar&Andreou 98], Speaker Adaptation (such as MLLR [Legetter&Woodland 95] and MAP[Gauvin&Lee 94]). The use of those methods will definitely increase the complexity of the decoding engine and make its speed slower than using relatively simple acoustic models as in this work. More techniques may be necessary to further speed up TigerEngine for accurate decoding of spontaneous speech.

Word-lattice/graph and multipass search. Currently TigerEngine 1.0 only supports one-pass and one-best decoding, but as described in Section 2.3, word-lattice/graph based

[23] Again, by using __int64 data type in Visual C++ 6.0, an initial new version of TigerEngine has already been built up to support large vocabulary size (larger than 64K).

multipass search are very useful in several ways: firstly, it is very useful for integrating recognition results obtained from different models with different degrees of complexity and precision; secondly, word-lattice/graph is very suitable for some post-processing tasks such as confusion network optimization [Mangu+ 00, Xue&Zhao 05] and speech understanding; thirdly, discriminative training such as Minimum Classification Error (MCE) [Chou+ 96] and MMIE (Maximum Mutual Information) [Woodland&Povey 02] methods need word-lattice/graph to provide competing hypotheses for each sentence. By extending TigerEngine's ability to provide word-lattice/graph, many new research topics will be enabled for our research group and it will benefit MCE or MMIE training for spontaneous speech recognition.

References

[Alleva+ 92] Alleva, F., Hon, Huang, X., 1992. Applying SPHINX-II to the DARPA Wall Street Journal CSR task. Proc. of DARPA Speech and Natural Language Workshop, pp.393–398, Harriman, NY, USA.

[Alleva+ 96] Alleva, F., Huang, X., Hwang, M.-Y., 1996. Improvements on the pronunciation prefix tree search organization. Proc. of ICASSP (IEEE International Conference on Acoustics, Speech, and Audio Processing), 1, pp.133-136, Atlanta, GA, USA.

[Alleva 97] Alleva, F., 1997. Search organization in the Whisper continuous speech recognition system. Proc. of IEEE Workshop on Automatic Speech Recognition and Understanding, pp.295-302.

[Atal&Schroeder 67] B.S. Atal and M. R. Schroeder, 1967. Predictive coding of speech signals. Proc. 1967 AFCRL/IEEE Conference on Speech Communication and Processing, pp. 360–361, Cambridge, Mass, 1967.

[Aubert 99] Aubert, X., 1999. One pass cross word decoding for large vocabularies based on a lexical tree search organization. Proc. of Eurospeech (European Conference on Speech Communication and Technology), Budapest, Hungary.

[Aubert 02] Aubert, X., 2002. An overview of decoding techniques for large vocabulary continuous speech recognition. *Computer Speech and Language*, 16, pp.89-114.

[Beulen+ 99] Beulen, K., Ortmanns, S., Elting, C., 1999. Dynamic Programming Search Techniques for Crossword Modeling in Speech Recognition. Proc. of ICASSP, pp.609–612, Phoenix, AZ, USA, March.

[Bahl+ 83] Bahl, L., Jelinek, F., Mercer, R., 1983. A Maximum Likelihood Approach to Continuous Speech Recognition. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 20, pp.179-190.

[Bahl+ 87] Bahl, L., et al, 1987. Speech Recognition with Continuous-Parameter Hidden Markov Models, *Computer Speech and Language*, 2, pp.219-234.

[Bahl+ 89] Bahl, L., et al., 1989. Large Vocabulary Natural Language Continuous Speech Recognition. Proc. of ICASSP, pp.465-467, Glasgow, Scotland, UK.

[Baker 87] Baker, J., 1987. The DRAGON System – An Overview. *IEEE Trans. Acoustics, Speech and Signal Processing*, 23 (1), pp.24-29.

[Baum 72] Baum, L., 1972. An inequalityand associated maximization technique in statistical estimation for probabilistic functions of Markov processes," *Inequalities*, 3, pp.1-8.

[Bellman 57] Bellman, R., 1957. **Dynamic Programming.** Princeton University Press.

[Bilmes&Zweig 02] Bilmes, J. and Zweig, G., 2002. The graphical models toolkit: an open source software system for speech and time-series processing. Proc. of ICASSP, 4, pp.3916–3919, Orlando, FL, USA.

[Bocchiere 93] Bocchiere, E., 1993. Vector Quantization for efficient computation of continuous density likelihood. Proc. of ICASSP, 2, pp.682-695, Minneapolis, Minnesota, USA.

[Bryant&O'Hallaron 03] Bryant, R. E. and O'Hallaron, D. R., 2003. **Computer Systems - A Programmer's Perspective.** Pearson Education, Inc.

[Cardenal+ 02] Cardenal-Lopez, A., Javier Dieguez-Tirado A., Garcia-Mateo, C., 2002. Fast LM look-ahead for large vocabulary continuous speech recognition using perfect hashing. Proc. of ICASSP, 1, pp.705 –I708, Orlando, FL, USA.

[Chen&Goodman 98] Chen, S. and Goodman, J., 1998. An empirical study of smoothing techniques for language modeling. Technical Report, TR-10-98, Center for Research in Computing Technology, Harvard University.

[Chou+ 96] Chou, W., Juang, B., and Lee, C., 1996. A minimum error rate pattern recognition approach to speech recognition. *International Journal on Pattern Recognition and Artificial Intelligence*, 8, pp.5-31.

[Chou+ 00] Chou, W., Juang, B., and Lee, C., 2000. Discriminant-function-based minimum recognition error rate pattern-recognition approach to speech recognition. *Proceedings of the IEEE*, 88(8), pp.1201-1223.

[Church&Gale 91] Church, K. and Gale, W., 1991. A comparison of the enhanced Good-Turing and deleted estimation methods for estimating probabilities of English bigrams. *Computer Speech and Language*, 5(1), pp.19-54.

[Cormen+ 01] Cormen, T., Leiserson, C., Rivest, R., Stein, C., 2001. **Introduction to Algorithms (Second Edition)**, MIT Press.

[Czech+ 92] Czech. Z., Havas, G., Majewski, B., 1992. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5), pp.257-264.

[Davis&Mermelsten 80] Davis, S. and Mermelsten, P., 1980. Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuous Spoken Sentences. *IEEE Trans. Acoustics, Speech and Signal Processing*, 28 (4), pp.357-366.

[Demuynck+ 02] Kemuynck, K., Compernolle, D., Wambacq, P., 2002. Doing away with the Viterbi approximiation. Proc. of ICASSP, 1, pp. 717-720, Orlando, FL, USA.

[Dempster+ 77] Dempster, A., Laird, N., and Rubin, D., 1977. Maximum likelihoood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 39(B), pp.1–38.

[Deshmukh+ 99] Deshmukh, N., Ganapathiraju, A., Picone, J., 1999. Hierarchical search – working towards a solution to the decoding problem. *IEEE Signal Processing Magazine*, 16 (5), pp.84-107.

[Deng&O'Shaughnessy 03] Deng, L., O'Shaughnessy, D., 2003. **Speech Processing – A Dynamic and Optimization-Oriented Approach,** Marcel Dekker, Inc.

[Evermann+ 05] Evermann, G., Chan, H., Gales, M., Jia, B., Marva, D., Woodland, P., Yu, K., 2005. Training LVCSR systems on thousands of hours of data. Proc. of ICASSP, 1, pp.209 ~ 212, Philadelphia, PA, USA.

[Fiscus 97] Fiscus, J., 1997. A post-processing system to yield reduced word error rates: recognizer output voting error reduction (ROVER). Proc. of IEEE Workshop on Automatic Speech Recognition and Understanding, Santa Barbara, CA, pp.347-354.

[Fox+ 92] Fox, E., Heath, L., Chen, Q., and Daoud, A., 1992. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1), pp.105-121.

[Gauvain&Lee 94] Gauvain, J. and C.-H. Lee, 1994. Maximum A Posteriori Estimation for Multivariate Gaussian Mixture Observations of Markov Chains. *IEEE Trans. Speech and Audio Processing*, 2(2), pp.291-298.

[Gopalakrishnan+ 95] Gopalakrishnan, P., Bahl, L., Mercer, R., 1995. A tree search strategy for Large-Vocabulary Continuous Speech Recognition. Proc. of ICASSP, pp.572-575, Detroit, MI, USA.

[Hermansky 90] Hermansky, H., 1990. Perceptual Linear Predictive (PLP) Analysis of Speech. *Journal of Acoust. Soc. Am.*, pp.1738-1752.

[Huang+ 01] Huang, X., Acero, A., Hon, H., 2001. **Spoken Language Processing**. Prentice Hall Press.

[Hwang+ 89] Hwang, M., Hon, H., Lee, K., 1989. Modeling Between-Word Coarticulation in Continuous Speech Recognition. Proc.of Eurospeech, pp.5–8, Paris, France.

[Hwang&Huang 92] Hwang, M. and Huang, X., 1992. Subphonetic Modeling with Markov States-Senone. Proc. of ICASSP, 1, pp.33-36, San Francisco, CA, USA.

[Hwang+ 93] Hwang, M., Huang, X., and Alleva, F., 1993. Predicting Unseen Triphones with Senones. Proc. of ICASSP, 2, pp.311-314, Minneapolis, Minnesota, USA.

[Jelinek 91] Jelinek, F., 1991. Up from trigrams! The struggle for improved language models. Proc. of Eurospeech, pp.1037-1040, Genoa, Italy.

[Jenkins 95] Jenkins, R., 1995. Hash Functions for Hash Table Lookup. http://burtleburtle.net/bob/hash/evahash.html.

[Juang&Rabiner 90] Juang, B. and Rabiner, L., 1990. The segmenatal k-means algorithm for estimating parameters of Hidden Markov Models. *IEEE Trans. Speech Audio Processing*, 38(9), pp.1639-1641.

[Katz 87] S.M. Katz, 1987. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Trans. on Acoustics, Speech and Signal Processing*, Vol. ASSP-35, pp. 400-401, March.

[Kneser&Ney 95] Kneser, R. and Ney, H., 1995. Improved backing-off for M-gram language modeling. Proc. of ICASSP, pp.181-184, Detroit, MI, USA.

[Kumar&Andreou 98] Kumar, N. and Andreou, 1998. A. Heteroscedastic discriminant analysis and reduced rank HMMs for improved speech recognition. *Speech Communication*, 26, pp.283–297.

[Lari&Young 91] Lari, K. and Young, S., 1991. Applications of Stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 5(3), pp.237-257.

[Lavie+ 97] Lavie, A., Waibel, A., et al., 1997. Janus-III: Speech-to-Speech Translation in Multiple Languages, Proc. of ICASSP, pp.21-24, Munich, Germany.

[Lee+ 89] Lee, K., Hon, H., Hwang, M., Mahajan, M., Reddy, M., 1989. The SPHINX Speech Recognition System. Proc. of ICASSP, pp.445–448, Glasgow, Scotland, UK.

[Legetter & Woodland 95] Legetter, C., Woodland, P., 1995. Maximum likelihood linear regression for speaker sdaptation of continuous eensity hidden markov models.*Computer Speech and Language*, 9, pp.171–185.

[Li 03] Li, K., 2003. *Assembly Optimization of Speech Recognition System*. Master thesis, Computer Science department, Univ. of Missouri-Columbia.

[Li&Zhao 03] Li, X., Zhao, Y., 2003. Exploiting order-preserving minimum perfect hashing to speedup N-gram language model look-ahead. Proc. of Eurospeech, pp.2281-2284, Geneva, Switzerland.

[Li&Zhao 05] Li, X., Zhao, Y., 2005. A fast and memory-efficient N-gram language model lookup method for Large Vocabulary Continuous Speech Recognition. Accepted to *Computer Speech and Language*.

[Lowerre 76] Lowerre, B., 1976. *The HARPY Speech Recognition System*. PhD Dissertation in Computer Science Department, Carnegie Mellon University.

[Mangu+ 00] Mangu, L., Brill, E., and A. Stolcke, 2000. Finding consensus in speech recognition: word error minization and other application of confusion network. *Computer Speech and Language*, 14 (4), pp.373-400.

[Makhoul 75] Makhoul, J., 1975. Linear prediction: A tutorial review. *Proceedings of the IEEE*, 63(4), pp.561-580.

[Martin+ 01] Martin, A., Przybocki, M., 2001. The 2001 NIST evaluation for recognition of conversational speech over the telephone. Proc. of NIST Large Vocabulary Conversational Speech Recognition Workshop, MD, USA.

[Mohri+ 02] M. Mohri , F. Pereira and M. Riley, 2002. Weighted Finite-State

Transducers in Speech Recognition. *Computer Speech & Language*, Volume 16, Issue 1, January 2002, Pages 69-88.

[Morris+ 00] Morris, A., Payne, Josifovski, L., Bourlard, H., Cooke, M., Gree, P., 2000. A neural network for classification with incomplete data: application to robust ASR. Proc. of ICSLP (International Conference on Spoken Language Processing), pp.345-348, Beijing, China.

[Ney+ 87] Ney, H., Mergel, D., Noll, A., Paeseler, A., 1987. A data-driven organization of the dynamic processing beam search for continuous speech recognition. Proc. of ICASSP, pp.833-836, Dallas, TX, USA.

[Ney+ 92] Ney, H., Haeb-Umbach, R., Tran, B., Oerder, M., 1992. Improvements in Beam Search for 10000-word continuous speech recognition. Proc. of ICASSP, pp.9-12, San Francisco, CA, USA.

[Ney&Ortmanns 99] Ney, H., Ortmanns, S., 1999. Dynamic programming search for continuous speech recognition. *IEEE Signal Processing Magazine*, 16 (5), pp.64-83.

[Nilsson 98] Nilsson, N.J., 1998. **Artificial Intelligence: A New Synthesis.** Academic Press/ Morgan Kaufmann.

[Ney+ 98] Ney, H., Welling, L., Ortmanns, S., Beulen, K., Wessel, F., 1998. RWTH large vocabulary continuous speech recognition system. Proc. of ICASSP, pp.853-856, Seattle, WA, USA.

[Odell+ 94] Odell, J., Valtchev, V., Woodland, P., Young, S., 1994. A One-Pass Decoder Design for Large Vocabulary Recognition. Proc. ARPA Spoken Language Technology Workshop, pp.405–410, Plainsboro, NJ, USA.

[Odell 95] Odell, J., 1995. *The Use of Context in Large Vocabulary Speech Recognition*. Ph.D. thesis, University of Cambridge, Cambridge, UK.

[Ostendorf&Roukos 89] Ostendorf, M. and Roukos, S., 1989. A stochastic segment model for phoneme-based continuous speech recognition. *IEEE Trans. Acoustics, Speech and Signal Processing*, 37 (1), pp.1957-1869.

[Ortmanns+ 96a] Ortmanns, S., Ney, H., Eiden, A., 1996. Language-model look-ahead for large vocabulary speech recognition. Proc. of ICSLP, 4, pp.2095-2098, Philadelphia, PA, USA.

[Ortmanns+ 96b] Ortmanns, S., Ney, H., Seide, F., Lindam, I., 1996. A Comparison of Time Conditioned and Word Conditioned Search Techniques for Large Vocabulary Speech Recognition. Proc of ICASSP, 4, pp.2091–2094, Philadelphia, PA, USA.

[Ortmanns+ 97a] Ortmanns, S., Ney, H., Aubert, X., 1997. A Word graph algorithm for large vocabulary continuous speech recognition. *Computer Speech and Language*, 11(1), pp.43-72.

[Ortmanns+ 97b] Ortmanns, S., Eiden, A., Ney, H., Coenen, N., 1997. Look-ahead Techniques for Fast Beam Search. Proc.of ICASSP, pp.1783-1786, Munich, Germany.

[Ortmanns+ 98] Ortmanns, S., Eiden, A., Ney, H., 1998. Improved Lexical Tree Search for large vocabulary speech recognition. Proc. of ICASSP, pp.817-820.

[Ortmanns+ 99] Ortmanns, S., Reichl, W., Chou, W., 1999. An Efficient Decoding Method for Real Time Speech Recognition. Proc. of Eurospeech, pp.499–502, Budapest, Hungary.

[Reichl&Chou 00] Reichl, w. and chou, w., 2000. Robust decision tree state tying for continuous speech recognition. *IEEE Trans. Speech and Audio Processing*, 8(5), pp.555-566.

[Pallett 02] Pallet, D., 2002. The role of the NIST in DARPA’s broadcast news continuous speech recognition research program. *Speech Communication*, 37, pp.3-14.

[Rabiner 89] Rabiner, L., 1989. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, *Proceedings of the IEEE*, 77(2), pp.257-286.

[Rabiner&Juang 93] Rabiner, L. R., and Juang, Biing-Hwang, 1993. **Fundamentals of Speech Recognition.** Prentice Hall Press.

[Schwartz+ 85] Schwartz, R., Chow, Y., Kimball, O., Roucos, S., Krasner, M., Makhoul, J., 1985. Context-Dependent Modeling for Acoustic-Phonetic Recognition of Speech Signals. Proc. of ICASSP, pp.1205-1208, Berlin, Germany.

[Schwartz+ 92] Schwartz, R., Austin, S., et. al., 1992. New Uses For the N-Best Sentence Hypotheses Within the Byblos Speech Recognition System. Proc. of ICASSP, 1, pp.1–4, San Francisco, CA, USA.

[Sixtus&Ney 02] Sixtus, A., and Ney, H., 2002. From within-word model search to

crossword model search in large vocabulary continuous speech recognition. *Computer Speech and Language*, 16(2), pp.245-271.

[Sixtus+ 00] Sixtus, A., Molau, S., Kanthak, S., Schl¨uter, R., Ney, H., 2000. Recent Improvements of the RWTH Large Vocabulary Speech Recognition System on Spontaneous Speech. Proc. of ICASSP, 3, pp. 1671–1674, Istanbul, Turkey, June 2000.

[Sixtus 03] Sixtus, A., 2003. *Crossword Phoneme Models for Large Vocabulary Continuous Speech Recognition*. Ph.D. dissertation, RWTH, Germany.

[Slotau+ 02] Slotau, H., Metze, F., Fugen, C., Waibel, A., 2002. Efficient language model lookahead through polymorphic linguistic context assignment. Proc. of ICASSP, 1, pp.709-712, Orlando, FL, USA.

[Steinbiss & Tran+ 94] Steinbiss, V., Tran, B., Ney, H., 1994. Improvements in Beam Search. Proc. Proc. of *International Conference on Spoken Language Processing*, Vol. 4, pp.2143–2146, Yokohama, Japan.

[Stolcke 02] Stolcke, A., 2002. SRILM – An extensible language modeling toolkit. Proc. of ICSLP, 2, pp.901-904, Denver, CO, USA.

[Stolcke+ 00] Stolcke, A., Bratt, H., Butzberger, J., Franco, H., Gadde, V., Plauch, M., Richey, C., Shriberg, E., Somez, K., Weng, F., and Zheng, J., 2000. The SRI March 2000 Hub-5 conversational speech transcription System. Proc. of NIST Speech Transcription Workshop, College Park, MD, USA.

[Sundaram+ 01] Sundaram, R., Hamaker, J., Picone, J., 2001. Twister: the ISIP 2001 conversational speech evaluation system. Proc. of NIST Large Vocabulary Conversational Speech Recognition Workshop, MD, USA.

[Tarjan 83] Tarjan, R., 1983. **Data Structures and Network Algorithms**. SIAM Monograph #44.

[Tomita 97] Tomita, M., 1987. An efficient augmented-context-free parsing algorithm. *Computer Linguistics*, 13 (1-2), pp.31-46.

[Intel 05] Intel’s website, 2005. Getting Started with SSE/SSE2 for the Intel® Pentium® 4 Processor. http://www.intel.com/cd/ids/developer/asmo-na/eng/events/newsletter/20240.htm.

[Wakita 77] Wakita, H., 1977. Normalization of vowels by vocal tract length and its application to vowel identification. *IEEE Trans. Acoustics, Speech, and Signal Processing*, 25 (2), pp.183–192.

[Woodland+ 95] Woodland, P., Leggetter, C., Odell, J., Valtchev, V., and Young, S., 1995. The 1994 HTK large vocabulary speech recognition system. Proc. of ICASSP, 1, pp.73-76, Detroit, MI, USA.

[Woodland+ 02] Woodland, P., Evermann, G., Gales, M., Hain, T., Liu, A., Moore,G., Povey, D., Wang, L., 2002. CU-HTK April 2002 Switchboard System. Proc. of NIST Rich Transcription Workshop, Vienna, VA, USA.

[Woodland & Povey 02] Woodland, P., Povey, D., 2002. Large scale discriminative training of Hidden Markov Models for speech recognition, *Computer Speech and Language*, 16 (1), pp.25–47.

[Woszczyna 98] Woszczyna, M., 1998. *Fast Speaker Independent Large Vocabulary Continuous Speech Recognition*. PhD Dissertation, Computer Science Department, Carnegie Mellon University.

[Xue&Zhao 05] Xue, J. and Zhao, Y., 2005. Improved confusion network algorithm and shortest path search from word lattice. Proc. of ICASSP, 1, pp.853~856, Philadelphia, PA, USA.

[Young+ 89] Young, S., Russel, R., Thornton, J., 1989. Token Passing: A simple conceptual model for connected speech recognition systems. Technical Report, Cambridge University Engineering Department.

[Young+ 00] Young, S., Kershaw, D., Odell, J., Allison, D., Valtchev, V., Woodland, P., 2000. **The HTK Book Version 3.0**. Cambridge, England, Cambridge University.

[Zhang&Zhao 02] Zhang, X., Zhao Y., 2002. Minimum perfect hashing for fast N-gram language model lookup. Proc. of ICSLP, 1, pp.401- 404, Denver, CO, USA.

Vita

Xiaolong Li was born on March 21, 1976, in Shaoyang, Hunan, China. After attending public schools in Hunan, China, he received the following degrees: B.S. in Electrical Engineering from University of Science and Technology of China (USTC) at Hefei, Anhui, China (1998); M.E. in Signal and Information Processing from Peking University at Beijing, China (2001); Ph.D. in Computer Science from University of Missouri at Columbia, Missouri, USA (2005). He is married to Wei Qiao of Qiqihar, Heilongjiang, China, and he is presently a member of the Advanced Technology Incubation Group at Microsoft, Redmond, Washington, USA.