THE IBM ATTILA SPEECH RECOGNITION TOOLKIT

Hagen Soltau, George Saon, and Brian Kingsbury

IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA
{hsoltau,gsaon,bedk}@us.ibm.com

ABSTRACT

We describe the design of IBM's Attila speech recognition toolkit. We show how the combination of a highly modular and efficient library of low-level C++ classes with simple interfaces, an interconnection layer implemented in a modern scripting language (Python), and a standardized collection of scripts for system-building produce a flexible and scalable toolkit that is useful both for basic research and for construction of large transcription systems for competitive evaluations.

Index Terms: speech recognition

1. INTRODUCTION

Our goals for the Attila toolkit were driven by our previous experience using other toolkits for both basic research and construction of large evaluation systems. A key to successful evaluation systems, for example in the DARPA EARS and GALE programs, is completing a large number of experiments in a short amount of time: efficient implementation and scalability to large compute clusters are crucial. A key to success in basic research is rapidly prototyping new ideas without needing to write a lot of low-level code: a researcher should be able to focus on the algorithm without needing to satisfy complex interfaces. In summary, the design of the toolkit is based on the following wish list:

Flexibility A rich interface that supports fast prototyping.

Efficiency Minimal overhead for fast experiment turnaround.

Simplicity A focus on the algorithm, not on the interface code.

Maintainability A small, low-complexity code base.

The last goal is motivated by our observation that some automatic speech recognition toolkits, including a previous internal toolkit, comprise hundreds of thousands of lines of code, leading new users to duplicate preexisting functions simply because they could not comprehend the existing code.

2. DESIGN

To accomplish these goals, the toolkit makes a clear distinction between core algorithms and glue code by combining the advantages of a high-level scripting language with the efficiency of C++ [1]. Traditionally, C++ modules are assembled into executables that are controlled via a command-line interface, configuration files, or both, and the executables are managed from the scripting language. An example of this approach is HTK. This approach is cumbersome because it entails the parsing of many parameters and provides only very coarse-grained access to the underlying C++ classes. We opt for a different approach, in which the target scripting language, Python, is extended with customized C++ classes. A key enabler is `SWIG` [2], a tool that automatically generates interface code from the header files of a C++ library. We have designed the C++ classes in Attila such that most class members are public; thus, nearly all C++ data structures are exposed within Python.

Attila consists of three layers.

1. The **C++ Library**, which contains all the low-level classes needed to build a modern, large-vocabulary acoustic model, as well as two different decoders.

2. The **Python Library**, which contains modules that connect objects from the C++ library together into higher-level building blocks used to train and test speech recognition systems.

3. The **Attila Training Recipe (ATR)**: a collection of standard scripts to build state-of-the-art large vocabulary speech recognition systems. These scripts allow even inexperienced users to build new systems from scratch, starting with a flat-start procedure and culminating in discriminatively trained, speaker-adaptive models.

Figure 1 illustrates the structure of the toolkit. The C++ classes are represented by proxy Python classes that are generated automatically using `SWIG`. The modules in the Python library provide the glue to create higher-level functions used in scripts for training and decoding. The main benefit of this design is that it offers maximal flexibility without requiring the writing of interface code and without sacrificing effi-

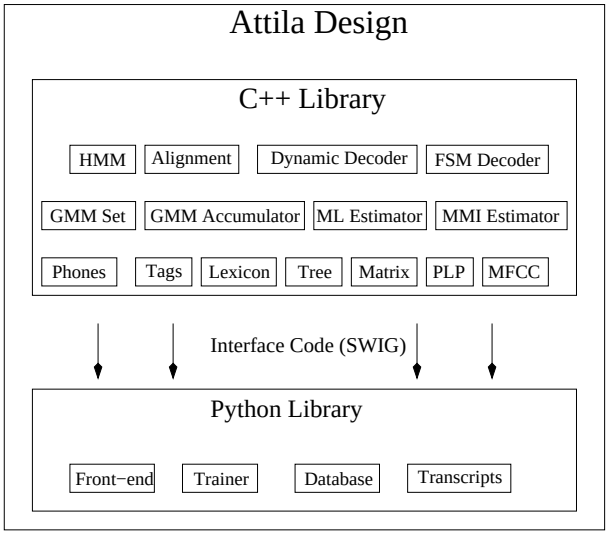

Fig. 1. Structure of the Attila toolkit.

ciency. In the next paragraphs we highlight some of the design choices we made that we found particularly useful.

2.1. Separation of Models, Accumulators, and Estimators

As shown in Figure 1, we have separate objects for models (e.g., Gaussian mixture models), accumulators (to hold sufficient statistics), and estimators (e.g., maximum likelihood and maximum mutual information). This allows us to reuse components and combine them in new ways. For example, the accumulator routines can be used for both maximum likelihood (ML) and maximum mutual information (MMI) estimation. While the ML estimator uses only one accumulator, the MMI estimators will update the models using two accumulators, one for the numerator statistics and one for the denominator statistics.

2.2. Abstraction of Alignments

Our alignment object is simply a container holding a set of hidden Markov model (HMM) state posterior probabilities for each frame. An alignment can be populated using several different methods: Viterbi, Baum-Welch, modified forward-backward routines over lattices for minimum phone error (MPE) or boosted MMI training, uniform segmentation for flat-start training, or conversion of manual labels as in the TIMIT corpus. Accumulator objects accumulate sufficient statistics only through an alignment object. This makes it easy to add new models to the toolkit, because only a method to accumulate sufficient statistics given an alignment and a method to update the model parameters from the statistics are required. Likewise, it is easy to add new alignment methods, because the designer only needs to worry about properly populating the alignment object.

2.3. Acoustic Scorer Interface

Because we are interested in working with a variety of acoustic models, including Gaussian mixture models, neural networks, and exponential models, we use an abstract interface for acoustic scoring that permits us to use any acoustic model with any decoder or alignment method. The interface consists of only a few functions:

```
class Scorer:
virtual int   get_model_count()=0;
virtual int   get_frame_count()=0;
virtual int   set_frame(int frameX)=0;
virtual float get_score(int modelX)=0;
```

`get_model_count` returns the number of models in a model container (e.g., the number of HMM states). `get_frame_count` returns the number of available frames in the current utterance. `set_frame` selects a frame for scoring. `get_score` returns the score (scaled negative log-likelihood) for the `modelX`-th model for the current frame.

2.4. Language Model Interface

Because the toolkit provides two different decoders, a static FSM decoder and a dynamic network decoder, and several different types of language model, it was necessary to use an abstract language model interface to maximize interoperability.

```
class LM:
virtual STATE start (WORD wordX);
virtual STATE extend(STATE state, WORD wordX)=0;
virtual SCORE score (STATE state, WORD wordX)=0;
virtual void  score (STATE state, SCORE *scoreptr)=0;
```

The language model state is an abstraction of the n-gram history that provides a unified view of the language model. The decoder accesses the language model only through `LM::STATE` instances. At the start of the utterance, the decoder generates an initial language model state by calling `start(wordX)`. The state is updated by calling `extend(state,wordX)` when transitioning to a new word. Decoding with n-gram models or finite state grammars can be easily expressed with this interface, as can lattice rescoring. The second variant of the `score` method retrieves the language model scores for all words in the vocabulary for a given `state`. This function is needed for fast language model access when computing lookahead scores for the dynamic network decoder.

2.5. Front End Interface

We implement dependency resolution for speakers and utterances in the Python layer, within a base class for all front end classes, and allow front end module instances to depend on the outputs of multiple other instances. We also use a very simple interface for front end modules: all modules produce matrices of floating-point numbers, and, with the exception of the module responsible for audio input, all modules accept matrices of floats as input. These two features have important consequences. First, authors of the front end C++ classes

can focus solely on the algorithmic details, and do not need to worry about how their code will interact with other classes: as long as they can accept and produce matrices of floats, the Python library will handle the rest. Second, we can realize front end signal processing algorithms as directed acyclic graphs of interacting instances, allowing for the production, for example, of perceptual linear prediction (PLP) and pitch features in a Mandarin speech recognition system.

3. CAPABILITIES

3.1. Front End

Available front end modules include audio input from a file, audio input from a socket, audio input from a sound card, downsampling, power spectrum, Mel binning with support for vocal tract length normalization (VTLN), Gaussianization, PLP coefficients, Mel-frequency cepstral (MFCC) coefficients, mean and variance normalization, splicing of successive frames into supervectors, application of a linear transform or projection such as linear discriminant analysis (LDA), feature-space maximum likelihood linear regression (FMLLR), fusion of parallel feature streams, pitch estimation, and feature-space discriminative transforms.

3.2. Hidden Markov Models and Context Modifiers

We use a three layer structure for hidden Markov models: (1) word graph, (2) phone graph, and (3) state graph. The word graph is usually constructed using a "sausage" structure to represent pronunciation variants and optional words. During training, a linear word sequence is constructed from the reference, and then it is extended to a sausage by adding alternative pronunciations and marking some tokens (e.g., silence) as optional. The phone graph is generated by applying the pronunciation lexicon to the word graph. In a similar fashion, the state graph is generated from the phone graph by applying a user defined HMM topology. Users can manipulate the HMM directly at the scripting level, adding nodes and transitions, setting transition costs, and so on.

The HMM object also handles the phonetic context needed to build decision trees by making the phone graph *context dependent*. We use a token-passing scheme to induce context in the graph structure. A forward pass propagates left context, while a backward pass adds the right context. At each node, a vector representing the context is updated with the current phone and shifted to respect constraints on the extent of the context. A new copy of the node is created for each unique context. This algorithm is surprisingly simple, and allows the use of long-range within-word and across-word contexts. The context in the HMM phone graph encodes both phone identity and position within the word.

To make the HMM context handling even more flexible, we use an abstract interface to define `ContextModifier` objects that transform the HMM-induced context by adding additional constraints. A `ContextModifier` is used by attaching it to a decision tree or decoding graph constructor. We have used this mechanism to limit the extent of cross-word context for small-footprint applications, and to combine acoustic models with different context configurations using tree arrays.

3.3. Decision Trees and Tree Arrays

To build decision trees, we model untied, fully context-expanded HMM states using single diagonal-covariance Gaussians, and construct the decision trees in a greedy fashion, selecting the question that gives the best gain in likelihood. For large contexts and large training corpora, we must handle hundreds of millions of untied HMM states. For example, a vowelized model for our 2009 GALE Arabic system had about 78 million untied states. We use a memory manager to reduce fragmentation caused by millions of small objects, and we hash the HMM state context to optimize access to the untied states. The clustering procedure can ask questions about the identity of the center phone, which lets us share models between different phones. This is useful when the phone set is not well designed or when there is a very limited amount of training data.

Another feature in our toolkit is tree arrays: a mechanism for combining multiple acoustic models with different decision trees into a single acoustic model. The decision trees may have entirely different question sets and, with the use of context modifiers, different context sizes or phone sets. A tree array is basically a hash table mapping from tuples of states from the different decision trees to virtual model indices, and is generated during the construction of a decoding graph. The virtual model indices are stored in the decoding graph. At run time, the acoustic scorer translates this virtual model back into a set of physical models and combines the acoustic scores of the underlying models using a weighted sum. One application of tree arrays is audio-visual speech recognition. With tree arrays, the acoustic model can specialize for phonemes, while the visual model specializes for visemes, reducing the error rate by 3% on an audio-visual speech recognition task [3]. A second application uses tree arrays for system combination: an alternative to ROVER or cross-adaptation that requires only a single decoding pass.

3.4. Gaussian Mixture Models

Gaussian mixture models are split into two components: one container for mean and precision matrices, and a second for mixture weights. This allows us to build semi-continuous models, also called *Genone* models [4]. A semi-continuous model gave us an 0.5% improvement on top of our very best fully-continuous system for the 2009 Arabic GALE evaluation. The toolkit supports a variety of covariance structures: radial, diagonal, and full.

3.5. Neural Networks

The toolkit provides modules implementing feedforward, multilayer perceptrons, and both an acoustic scorer interface and a front end interface for neural networks. Thus, networks can be used both for acoustic modeling and for feature extraction. The core modules implement basic building blocks such as convolutional weights; logistic, hyperbolic tangent, and softmax nonlinearities; and squared error and cross-entropy loss functions. All modules can operate on and produce either matrices or 3-dimensional tensors. As in the front end, we use Python wrapper classes to manage the connection of these basic building blocks into networks, and we separate models, accumulators, and estimators into separate classes. Networks are trained using error backpropagation. Because the training routines are implemented as Python scripts, it is simple to perform either fully on-line or mini-batch updates of the network parameters. It is also possible to train networks using sequence-discriminative criteria such as boosted MMI or MPE [5]: a capability that was easily added because of the toolkit's modular structure and standardized interfaces.

3.6. Adaptation

Sufficient statistics for adaptation are accumulated based on the same alignment object used in acoustic model training. The toolkit supports three forms of feature-space adaptation:

1. Vocal tract length normalization (VTLN [6]). For the details of our VTLN implementation, see [7].
2. Feature-space maximum likelihood linear regression (FMLLR, also known as constrained MLLR [8]).
3. Gaussianization, a form of histogram normalization implemented using a table lookup of inverse Gaussian cumulative density function values [9].

Model-space adaptation is performed with multiple MLLR transforms [10] using a regression tree to generate transforms for different sets of mixture components. We also support mean and diagonal variance transform estimation under discriminative objective functions such as boosted MMI.

3.7. Discriminative Training

The toolkit supports a variety of techniques for discriminative training, operating in both feature and model space and using several criteria: maximum mutual information (MMI), minimum phone error [11] and boosted MMI [12]. The different objective functions are implemented using different forward-backward routines, but they all have the same interface: the alignment object. For model space training, posteriors in the alignment objects multiply the normalized component likelihoods in the accumulation of model sufficient statistics. The model update uses smoothing to the previous iteration's model and cancellation of common statistics

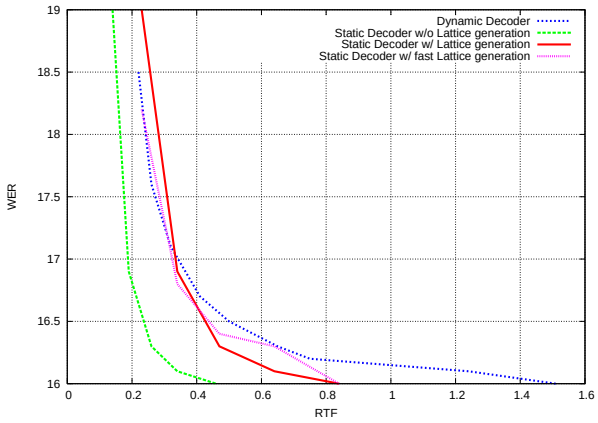

Fig. 2. English BN, SA models, 4.7m 4-gram LM

between the numerator and denominator paths, as explained in [12]. For feature space transformation estimation (e.g., fMPE [13]), the posteriors in the alignment objects are used to compute the gradient of the objective function with respect to the transformed features. This gradient is multiplied by the gradient of the transformed features with respect to either the main or the context transform to perform the model update. Competing paths (the denominator term) are encoded in word lattices generated by either of the decoders described below. We use a weak language model (unigram), and the language model states during decoding are given by a hashing of the entire word sequence up to the current frame [14]. We have observed gains from discriminative training of 10–30% relative over maximum-likelihood training, depending on the task and models.

3.8. Static Network Decoder

The static decoding graph can be constructed either through incremental expansion of the context decision tree [15], or through an incremental composition of the component finite state machines (FSMs) [16]. The FSM decoder, described in [14, 17], can produce a variety of outputs: 1-best hypotheses, word-level and HMM state-level lattices, and confusion networks. The decoder is very efficient, running in real time or faster without any significant loss in accuracy.

3.9. Dynamic Network Decoder

The dynamic network decoder [17] is a one-pass Viterbi decoder capable of handling large across-word contexts and large language model histories. The search network is based on a word loop and is fully minimized, including across-word context expansion. Word labels are shifted to allow for early path recombination. The language model lookahead uses the full language model history, and subtree dominance is exploited implicitly by a token pruning step. The Decoder can handle several types of the language model, including a new

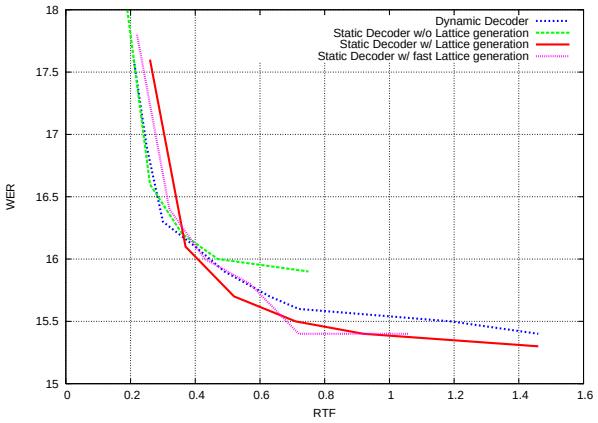

Fig. 3. English BN, SA models, 200m 4-gram LM

class based Maximum Entropy model described in [18]. The language model interface described earlier makes it easy to plug in new types of language models in the decoder without any need to modify the decoder itself.

A comparison of the decoder on an English Broadcast News task with a 90k vocabulary are shown in the figures 2 (small n-gram LM) and 3 (very large n-gram LM). Both decoders use the same acoustic and language models. The dynamic decoder uses the large n-gram LM directly, while the static decoder produces lattices first a small LM, followed by rescoring with the large LM. A more detailed discussion can be found in [17]. Decoding speed is measured as RTF (real time factor): processing time divided by segment duration. All experiments were run on a Intel Xeon 3.6GHz processor.

4. ATTILA TRAINING RECIPE

The Attila Training Recipe (ATR) is a set of standard scripts to build acoustic models. It begins with a flat-start procedure using an initial, uniform alignment of the HMM states and random sampling of pronunciations. This approach works well even for systems with many pronunciations, such as vowelized Arabic [19]. The uniform alignments initialize context-independent (CI) models, which are refined using the Baum-Welch algorithm. The CI models initialize a first set of context-dependent (CD) models, which are rebuilt several times, with the latter steps including speaker adaptation and discriminative training. Table 1 shows the performance of the recipe on a 50-hour English Broadcast News task. The ATR is the basis of all our evaluation systems, including English, Arabic and Mandarin broadcast, English and Spanish parliamentary [20], and conversational [7] tasks. Table 2 shows results for ASR systems that were trained on hundreds to thousands of hours using the toolkit and fielded in speech recognition evaluations [19, 21, 18]. As seen in recent GALE evaluations, the systems have excellent performance.

While the recipe was designed for large vocabulary tasks,

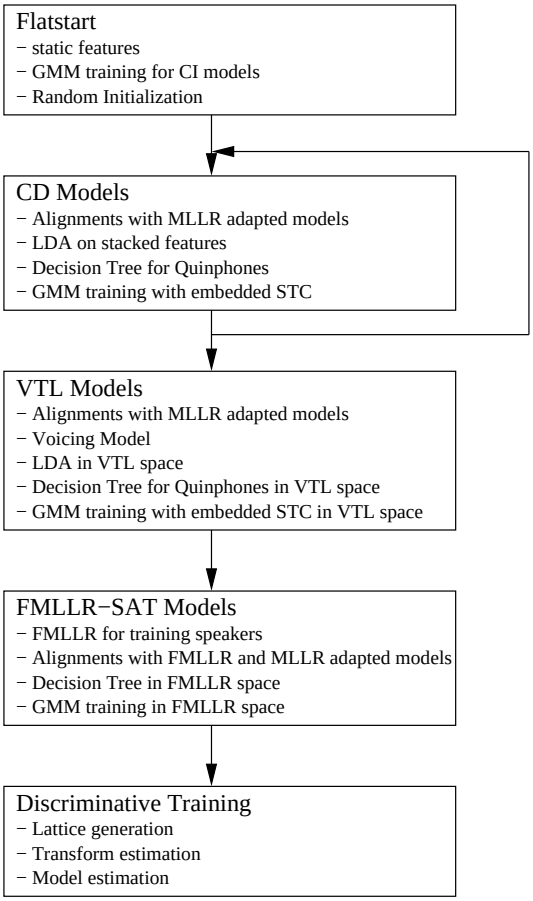

Fig. 4. Training Recipe. In each system building step, new alignments will be written using MLLR adapted models from the previous step. The estimation of a global semi-tied covariance (STC) is interleaved with the mixing-up procedure for the Gaussian mixture models (GMMs).

it also works very well on small tasks such as TIMIT, establishing a new baseline [22] for this task.

5. CONCLUSION

The combination of a carefully engineered library of C++ routines implementing core algorithms with a high-level scripting language is a good design for a speech recognition toolkit. While the toolkit has many modules including two full LVCSR decoders, the codebase itself is very small. The entire toolkit has only about $30,000$ lines of C++ code and about $5,000$ lines of Python code. The structure of the toolkit makes it easy to extend it, to add new training algorithms or incorporate new front-ends or other components.

The toolkit is very efficient and scalable, allowing us to build large transcription systems for competitive evaluations using thousands of hours of training data. At the same time, the toolkit is flexible and easy to use, making it a good en-

Step	WER
inital CD models	34.1%
retrained CD models	31.6%
retrained CD models	31.4%
VTLN models	25.6%
FMLLR-SAT models	23.1%
fMMI models	18.9%
fMMI+bMMI models	18.0%

Table 1. Performance of steps in the Attila Training Recipe

Task	Test	Performance
Arabic BN	EVAL'09	7.2% WER
Chinese BN	EVAL'08	9.1% CER
English BN	RT'04	12.3% WER

Table 2. Performance of evaluation systems trained on hundreds to thousands of hours of data.

vironment for basic research. For example, the toolkit has already been used in ways that we had not originally envisioned, such as training continuous space language models. Feedback from external academic partners who use the toolkit has also been very positive.

The toolkit is freely available to universities and government-related institutions and is distributed as binaries, header files, the Python interconnection layer and the ATR system building scripts. Please contact the authors for details.

We thank Dan Povey for his contributions to Attila. This work was partially supported by DARPA contract No. HR0011-06-2-0001. The views, opinions, and findings contained in this article are those of the authors, and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

6. REFERENCES

[1] J. K. Ousterhout, "Scripting: Higher level programming for the 21st century," *IEEE Computer*, vol. 31, no. 3, pp. 23–30, 1998.

[2] D. M. Beazley, "SWIG: An easy to use tool for integrating scripting languages with C and C++," in *Proc. Fourth Annual USENIX Tcl/Tk Workshop*, 1996.

[3] J. Huang and K. Visweswariah, "Improved decision trees for multi-stream HMM-based audio-visual continuous speech recognition," in *Proc. ASRU*, 2009, pp. 228–231.

[4] V. V. Digalakis, P. Monaco, and H. Murveit, "Genones: Generalized mixture tying in continuous hidden Markov model-based speech recognizers," *IEEE Trans. Speech and Audio Processing*, vol. 4, no. 4, pp. 281–289, 1996.

[5] B. Kingsbury, "Lattice-based optimization of sequence classification criteria for neural-network acoustic modeling," in *Proc. ICASSP*, 2009, pp. 3761–3764.

[6] S. Wegmann, D. McAllaster, J. Orloff, and B. Peskin, "Speaker normalization on conversational telephone speech," in *Proc. ICASSP*, 1996, vol. I, pp. 339–342.

[7] H. Soltau, B. Kingsbury, L. Mangu, D. Povey, G. Saon, and G. Zweig, "The IBM 2004 conversational telephony system for rich transcription," in *Proc. ICASSP*, 2005, vol. I, pp. 205–208.

[8] M. J. F. Gales, "Maximum-likelihood linear transforms for HMM-based speech recognition," *Computer Speech and Language*, vol. 12, no. 2, pp. 75–98, 1998.

[9] G. Saon, S. Dharanipragada, and D. Povey, "Feature space gaussianization," in *Proc. ICASSP*, 2004, vol. I, pp. 329–332.

[10] C. J. Leggetter and P. C. Woodland, "Speaker adaptation of continuous density HMMs using multivariate linear regression," in *Proc. ICSLP*, 1994.

[11] D. Povey and P. C. Woodland, "Minimum phone error and I-smoothing for improved discriminative training," in *Proc. ICASSP*, 2002, vol. I, pp. 105–108.

[12] D. Povey, D. Kanevsky, B. Kingsbury, B. Ramabhadran, G. Saon, and K. Visweswariah, "Boosted MMI for model and feature space discriminative training," in *Proc. ICASSP*, 2008, pp. 4057–4060.

[13] D. Povey, B. Kingsbury, L. Mangu, G. Saon, H. Soltau, and G. Zweig, "fMPE: Discriminatively trained features for speech recognition," in *Proc. ICASSP*, 2005, vol. I, pp. 961–964.

[14] G. Saon, D. Povey, and G. Zweig, "Anatomy of an extremely fast LVCSR decoder," in *Proc. Interspeech*, 2005, pp. 549–552.

[15] S. F. Chen, "Compiling large-context phonetic decision trees into finite-state transducers," in *Proc. Eurospeech*, 2003, pp. 1169–1172.

[16] M. Novak, "Incremental composition of static decoding graphs," in *Proc. Interspeech*, 2009, pp. 1175–1178.

[17] H. Soltau and G. Saon, "Dynamic network decoding revisited," in *Proc. ASRU*, 2009, pp. 276–281.

[18] S. F. Chen, L. Mangu, B. Ramabhadran, R. Sarikaya, and A. Sethy, "Scaling shrinkage-based language models," in *Proc. ASRU*, 2009, pp. 299–304.

[19] H. Soltau, G. Saon, B. Kingsbury, H.-K. J. Kuo, L. Mangu, D. Povey, and A. Emami, "Advances in Arabic speech transcription at IBM under the DARPA GALE program," *IEEE Trans. Audio, Speech, and Language Processing*, vol. 17, no. 5, pp. 884–894, 2009.

[20] B. Ramabhadran, O. Siohan, L. Mangu, G. Zweig, M. Westphal, H. Schulz, and A. Soneiro, "The IBM 2006 speech transcription system for European parliamentary speeches," in *Proc. Interspeech*, 2006, pp. 1225–1228.

[21] S. Chu, D. Povey, H.-K. Kuo, L. Mangu, S. Zhang, Q. Shi, and Y. Qin, "The 2009 IBM GALE Mandarin broadcast transcription system," in *Proc. ICASSP*, 2010, pp. 4374–4377.

[22] T. N. Sainath, B. Ramabhadran, and M. Picheny, "An exploration of large vocabulary tools for small vocabulary phonetic recognition," in *Proc. ASRU*, 2009, pp. 359–364.