Sphinx-4: A Flexible Open Source Framework for Speech Recognition

Willie Walker, Paul Lamere, Philip Kwok, Bhiksha Raj, Rita Singh, Evandro Gouvea, Peter Wolf, Joe Woelfel

Abstract

Sphinx-4 is a flexible, modular and pluggable framework to help foster new innovations in the core research of hidden Markov model (HMM) recognition systems. The design of Sphinx-4 is based on patterns that have emerged from the design of past systems as well as new requirements based on areas that researchers currently want to explore. To exercise this framework, and to provide researchers with a ”research-ready” system, Sphinx-4 also includes several implementations of both simple and state-of-the-art techniques. The framework and the implementations are all freely available via open source.

I. INTRODUCTION

WHEN researchers approach the problem of core speech recognition research, they are often faced with the problem of needing to develop an entire system from scratch, even if they only want to explore one facet of the field. Open source speech recognition systems are available, such as HTK [1], ISIP [2], AVCSR [3] and earlier versions of the Sphinx systems [4]–[6]. The available systems are typically optimized for a single approach to speech system design. As a result, these systems intrinsically create barriers to future research that departs from the original purpose of the system. In addition, some of these systems are encumbered by licensing agreements that make entry into the research arena difficult for non-academic institutions.

To facilitate new innovation in speech recognition research, we formed a distributed, cross-discipline team to create Sphinx-4 [7]: an open source platform that incorporates state-of-the art methodologies and also addresses the needs of emerging research areas. Given our technical goals as well as our diversity (e.g., we used different operating systems on different machines, etc.), we wrote Sphinx-4 in the Java^{TM}programming language, making it available to a large variety of development platforms.

First and foremost, Sphinx-4 is a modular and pluggable framework that incorporates design patterns from existing systems, with sufficient flexibility to support emerging areas of research interest. The framework is modular in that it comprises separable components dedicated to specific tasks, and it is pluggable in that modules can be easily replaced at runtime. To exercise the framework, and to provide researchers with a working system, Sphinx-4 also includes a variety of modules that implement state-of-the-art speech recognition techniques.

The remainder of this document describes the Sphinx-4 framework and implementation, and also includes a discussion of our experiences with Sphinx-4 to date.

II. SELECTED HISTORICAL SPEECH RECOGNITION SYSTEMS

The traditional approach to speech recognition system design has been to create an entire system optimized around a particular methodology. As evidenced by past research systems such as Dragon [8], Harpy [9], Sphinx and others, this approach has proved to be quite valuable in that the resulting systems have provided foundational methods for speech recognition research.

In the same light, however, each of these systems was largely dedicated to exploring a single specific groundbreaking area of speech recognition. For example, Baker introduced hidden Markov models (HMMs) with his Dragon system, [8], [10] and earlier predecessors of Sphinx explored variants of HMMs such as discrete HMMs [4], semicontinuous HMMs [5], and continuous HMMs [11]. Other systems explored specialized search strategies such as using lex tree searches for large N-Gram models [12].

Because they were focused on such fundamental core theories, the creators of these systems tended to hardwire their implementations to a high degree. For example, the predecessor Sphinx systems restrict the order of the HMMs to a constant value and also fix the unit context to a single left and right context. Sphinx-3 eliminated support for context free grammars (CFGs) due to the specialization on large N-Gram models. Furthermore, the decoding strategy of these systems tended to be deeply entangled with the rest of the system. As a result of these constraints, the systems were difficult to modify for experiments in other areas.

Design patterns for these systems emerged over time, however, as exemplified by Jelinek’s source-channel model [13] and Huang’s basic system architecture [14]. In developing Sphinx-4, one of our primary goals was to develop a framework that supported these design patterns, yet also allowed for experimentation in emerging areas of research.

W. Walker, P. Lamere, and P. Kwok are with Sun Microsystems
E. Gouvea and R. Singh are with Carnegie Mellon University
B. Raj, P, Wolf, and J. Woelfel are with Mitsubishi Electric Research Labs

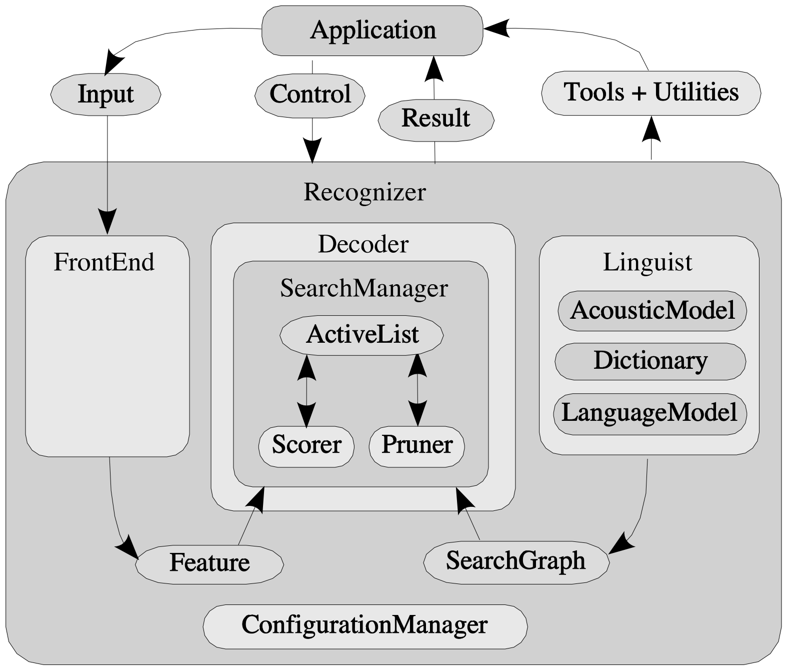

Fig. 1. Sphinx-4 Decoder Framework. The main blocks are the FrontEnd, the Decoder, and the Linguist. Supporting blocks include the ConfigurationManager and the Tools blocks. The communication between the blocks, as well as communication with an application, is depicted.

III. Sphinx-4 Framework

The Sphinx-4 framework has been designed with a high degree of flexibility and modularity. Figure 1 shows the overall architecture of the system. Each labeled element in Figure 1 represents a module that can be easily replaced, allowing researchers to experiment with different module implementations without needing to modify other portions of the system.

There are three primary modules in the Sphinx-4 framework: the *FrontEnd*, the *Decoder*, and the *Linguist*. The FrontEnd takes one or more input signals and parameterizes them into a sequence of *Features*. The Linguist translates any type of standard language model, along with pronunciation information from the *Dictionary* and structural information from one or more sets of *AcousticModels*, into a *SearchGraph*. The *SearchManager* in the Decoder uses the Features from the FrontEnd and the SearchGraph from the Linguist to perform the actual decoding, generating *Results*. At any time prior to or during the recognition process, the application can issue *Controls* to each of the modules, effectively becoming a partner in the recognition process.

The Sphinx-4 system is like most speech recognition systems in that it has a large number of configurable parameters, such as search beam size, for tuning the system performance. The Sphinx-4 *ConfigurationManager* is used to configure such parameters. Unlike other systems, however, the ConfigurationManager also gives Sphinx-4 the ability to dynamically load and configure modules at run time, yielding a flexible and pluggable system. For example, Sphinx-4 is typically configured with a FrontEnd (see Section IV) that produces Mel-Frequency Cepstral Coefficients (MFCCs) [15]. Using the ConfigurationManager, however, it is possible to reconfigure Sphinx-4 to construct a different FrontEnd that produces Perceptual Linear Prediction coefficients (PLP) [16] without needing to modify any source code or to recompile the system.

To give applications and developers the ability to track decoder statistics such as word error rate [17], runtime speed, and memory usage, Sphinx-4 provides a number of *Tools*. As with the rest of the system, the Tools are highly configurable, allowing users to perform a wide range of system analysis. Furthermore, the Tools also provides an interactive runtime environment that allows users to modify the parameters of the system while the system is running, allowing for rapid experimentation with various parameters settings.

Sphinx-4 also provides support for *Utilities* that support application-level processing of recognition results. For example, these utilities include support for obtaining result lattices, confidence scores, and natural language understanding.

IV. FrontEnd

The purpose of the FrontEnd is to parameterize an *Input* signal (e.g., audio) into a sequence of output *Features*. As illustrated in Figure 2, the FrontEnd comprises one or more parallel chains of replaceable communicating signal processing modules called *DataProcessors*. Supporting multiple chains permits simultaneous computation of different types of parameters from the same or different input signals. This enables the creation of systems that can simultaneously decode using different parameter types, such as MFCC and PLP, and even parameter types derived from non-speech signals such as video [3].

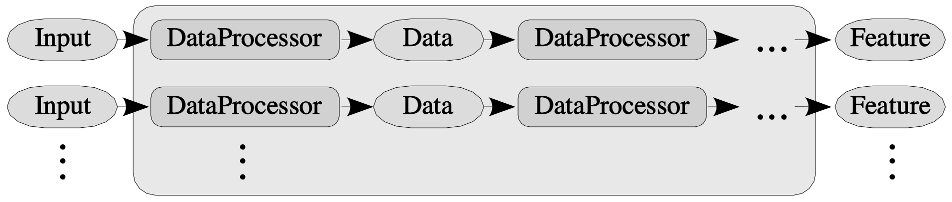

Fig. 2. Sphinx-4 FrontEnd. The FrontEnd comprises one or more parallel chains of communicating DataProcessors.

Like the ISIP [2] system, each DataProcessor in the FrontEnd provides an input and an output that can be connected to another DataProcessor, permitting arbitrarily long sequences of chains. The inputs and outputs of each DataProcessor are generic *Data* objects that encapsulate processed input data as well as markers that indicate data classification events such as end-point detection. The last DataProcessor in each chain is responsible for producing a Data object composed of parameterized signals, called *Features,* to be used by the Decoder.

Like the AVCSR system [3], Sphinx-4 permits the ability to produce parallel sequences of features. Sphinx-4 is unique, however, in that it allows for an arbitrary number of parallel streams.

The communication between blocks follows a pull design. With a pull design, a DataProcessor requests input from an earlier module only when needed, as opposed to the more conventional push design, where a module propagates its output to the succeeding module as soon as it is generated. This pull design enables the processors to perform buffering, allowing consumers to look forwards or backwards in time.

The ability to look forwards or backwards in time not only permits the Decoder to perform frame-synchronous Viterbi searches [18], but also allows the decoder to perform other types of searches such as depth-first and A* [19].

Within the generic FrontEnd framework, the Sphinx-4 provides a suite of DataProcessors that implement common signal processing techniques. These implementations include support for the following: reading from a variety of input formats for batch mode operation, reading from the system audio input device for live mode operation, preemphasis, windowing with a raised cosine transform (e.g., Hamming and Hanning windows), discrete fourier transform (via FFT), mel frequency filtering, bark frequency warping, discrete cosine transform (DCT), linear predictive encoding (LPC), end pointing, cepstral mean normalization (CMN), mel-cepstra frequency coefficient extraction (MFCC), and perceptual linear prediction coefficient extraction (PLP).

Using the ConfigurationManager described in Section III, users can chain the Sphinx-4 DataProcessors together in any manner as well as incorporate DataProcessor implementations of their own design. As such, the modular and pluggable nature of Sphinx-4 not only applies to the higher-level structure of Sphinx-4, but also applies to the higher-level modules themselves (i.e., the FrontEnd is a pluggable module, yet also consists of pluggable modules itself).

V. Linguist

The *Linguist* generates the SearchGraph that is used by the decoder during the search, while at the same time hiding the complexities involved in generating this graph. As is the case throughout Sphinx-4, the Linguist is a pluggable module, allowing people to dynamically configure the system with different Linguist implementations.

A typical Linguist implementation constructs the SearchGraph using the language structure as represented by a given LanguageModel and the topological structure of the AcousticModel (HMMs for the basic sound units used by the system). The Linguist may also use a Dictionary (typically a pronunciation lexicon) to map words from the LanguageModel into sequences of AcousticModel elements. When generating the SearchGraph, the Linguist may also incorporate sub-word units with contexts of arbitrary length, if provided.

By allowing different implementations of the Linguist to be plugged in at runtime, Sphinx-4 permits individuals to provide different configurations for different system and recognition requirements. For instance, a simple numerical digits recognition application might use a simple Linguist that keeps the search space entirely in memory. On the other hand, a dictation application with a 100K word vocabulary might use a sophisticated Linguist that keeps only a small portion of the potential search space in memory at a time.

The Linguist itself consists of three pluggable components: the LanguageModel, the Dictionary, and the AcousticModel, which are described in the following sections.

A. LanguageModel

The LanguageModel module of the Linguist provides word-level language structure, which can be represented by any number of pluggable implementations. These implementations typically fall into one of two categories: graph-driven grammars and stochastic N-Gram models. The graph-driven grammar represents a directed word graph where each node represents a single word and each arc represents the probability of a word transition taking place. The stochastic N-Gram models provide probabilities for words given the observation of the previous n-1 words.

The Sphinx-4 LanguageModel implementations support a variety of formats, including the following:

- `SimpleWordListGrammar`: defines a grammar based upon a list of words. An optional parameter defines whether the grammar "loops" or not. If the grammar does not loop, then the grammar will be used for isolated word recognition. If the grammar loops, then it will be used to support trivial connected word recognition that is the equivalent of a unigram grammar with equal probabilities.
- `JSGFGrammar`: supports the Java^{TM}Speech API Grammar Format (JSGF) [20], which defines a BNF-style, platform-independent, and vendor-independent Unicode representation of grammars.
- `LMGrammar`: defines a grammar based upon a statistical language model. LMGrammar generates one grammar node per word and works well with smaller unigram and bigram grammars of up to approximately 1000 words.
- `FSTGrammar`: supports a finite-state transducer (FST) [21] in the ARPA FST grammar format.
- `SimpleNGramModel`: provides support for ASCII N-Gram models in the ARPA format. The SimpleNGramModel makes no attempt to optimize memory usage, so it works best with small language models.
- `LargeTrigramModel`: provides support for true N-Gram models generated by the CMU-Cambridge Statistical Language Modeling Toolkit [22]. The LargeTrigramModel optimizes memory storage, allowing it to work with very large files of 100MB or more.

B. Dictionary

The *Dictionary* provides pronunications for words found in the LanguageModel. The pronunciations break words into sequences of sub-word units found in the AcousticModel. The Dictionary interface also supports the classification of words and allows for a single word to be in multiple classes.

Sphinx-4 currently provides implementions of the Dictionary interface to support the CMU Pronouncing Dictionary [23]. The various implementations optimize for usage patterns based on the size of the active vocabulary. For example, one implementation will load the entire vocabulary at system initialization time, whereas another implementation will only obtain pronunciations on demand.

C. AcousticModel

The *AcousticModel* module provides a mapping between a unit of speech and an HMM that can be scored against incoming features provided by the FrontEnd. As with other systems, the mapping may also take contextual and word position information into account. For example, in the case of triphones, the context represents the single phonemes to the left and right of the given phoneme, and the word position represents whether the triphone is at the beginning, middle, or end of a word (or is a word itself). The contextual definition is not fixed by Sphinx-4, allowing for the definition of AcousticModels that contain allophones as well as AcousticModels whose contexts do not need to be adjacent to the unit.

Typically, the Linguist breaks each word in the active vocabulary into a sequence of context-dependent sub-word units. The Linguist then passes the units and their contexts to the AcousticModel, retrieving the HMM graphs associated with those units. It then uses these HMM graphs in conjunction with the LanguageModel to construct the SearchGraph.

Unlike most speech recognition systems, which represent the HMM graphs as a fixed structure in memory, the Sphinx-4 HMM is merely a directed graph of objects. In this graph, each node corresponds to an HMM state and each arc represents the probability of transitioning from one state to another in the HMM. By representing the HMM as a directed graph of objects instead of a fixed structure, an implementation of the AcousticModel can easily supply HMMs with different topologies. For example, the AcousticModel interfaces do not restrict the HMMs in terms of the number of states, the number or transitions out of any state, or the direction of a transition (forward or backward). Furthermore, Sphinx-4 allows the number of states in an HMM to vary from one unit to another in the same AcousticModel.

Each HMM state is capable of producing a score from an observed feature. The actual code for computing the score is done by the HMM state itself, thus hiding its implementation from the rest of the system, even permitting differing probability density functions to be used per HMM state. The AcousticModel also allows sharing of various components at all levels. That is, the components that make up a particular HMM state such as Gaussian mixtures, transition matrices, and mixture weights can be shared by any of the HMM states to a very fine degree.

As with the rest of Sphinx-4, individuals can configure Sphinx-4 with different implementations of the AcousticModel based upon their needs. Sphinx-4 currently provides a single AcousticModel implementation that is capable of loading and using acoustic models generated by the Sphinx-3 trainer.

D. SearchGraph

Even though Linguists may be implemented in very different ways and the topologies of the search spaces generated by these Linguists can vary greatly, the search spaces are all represented as a SearchGraph. Illustrated by example in Figure 3, the SearchGraph is the primary data structure used during the decoding process.

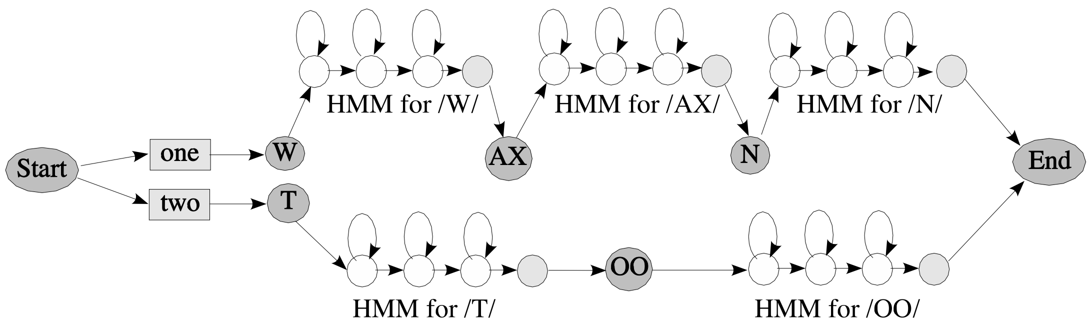

Fig. 3. Example SearchGraph. The SearchGraph is a directed graph composed of optionally emitting SearchStates and SearchStateArcs with transition probabilities. Each state in the graph can represent components from the LanguageModel (words in rectangles), Dictionary (sub-word units in dark circles) or AcousticModel (HMMs).

The graph is a directed graph in which each node, called a *SearchState*, represents either an *emitting* or a *non-emitting* state. Emitting states can be scored against incoming acoustic features while non-emitting states are generally used to represent higher-level linguistic constructs such as words and phonemes that are not directly scored against the incoming features. The arcs between states represent the possible state transitions, each of which has a probability representing the likelihood of transitioning along the arc.

The SearchGraph interface is purposely generic to allow for a wide range of implementation choices, relieving the assumptions and hard-wired constraints found in previous recognition systems. In particular, the Linguist places no inherent restrictions on the following:

- Overall search space topology
- Phonetic context size
- Type of grammar (stochastic or rule based)
- N-Gram language model depth

A key feature of the SearchGraph is that the implementation of the SearchState need not be fixed. As such, each Linguist implementation typically provides its own concrete implementation of the SearchState that can vary based upon the characteristics of the particular Linguist. For instance, a simple Linguist may provide an in-memory SearchGraph where each SearchState is simply a one-to-one mapping onto the nodes of the in-memory graph. A Linguist representing a very large and complex vocabulary, however, may build a compact internal representation of the SearchGraph. In this case, the Linguist would generate the set of successor SearchStates by dynamically expanding this compact representation on demand.

The manner in which the SearchGraph is constructed affects the memory footprint, speed, and recognition accuracy. The modularized design of Sphinx-4, however, allows different SearchGraph compilation strategies to be used without changing other aspects of the system. The choice between static and dynamic construction of language HMMs depends mainly on the vocabulary size, language model complexity and desired memory footprint of the system, and can be made by the application.

E. Implementations

As with the FrontEnd, Sphinx-4 provides several implementations of the Linguist to support different tasks.

The `FlatLinguist` is appropriate for recognition tasks that use context-free grammars (CFG), finite-state grammars (FSG), finite-state transducers (FST) and small N-Gram language models. The FlatLinguist converts any of these external language model formats into an internal Grammar structure. The Grammar represents a directed word graph where each *GrammarNode* represents a single word, and each arc in the graph represents the probability of a word transition taking place. The FlatLinguist generates the SearchGraph directly from this internal Grammar graph, storing the entire SearchGraph in memory. As such, the FlatLinguist is very fast, yet has difficulty handling grammars with high branching factors.

The `DynamicFlatLinguist` is similar to the FlatLinguist in that is is appropriate for similar recognition tasks. The main difference is that the DynamicFlatLinguist dynamically creates the SearchGraph on demand, giving it the capability to handle far more perplex grammars. With this capability, however, comes a cost of a modest decrease in run time performance.

The `LexTreeLinguist` is appropriate for large vocabulary recognition tasks that use large N-Gram language models. The order of the N-Grams is arbitrary, and the LexTreeLinguist will support true N-Gram decoding. The LexTreeLinguist organizes the words in a lex tree [6], a compact method of representing large vocabularies. The LexTreeLinguist uses this lex tree to dynamically generate SearchStates, enabling it to handle very large vocabularies using only a modest amount of memory. The LexTreeLinguist supports ASCII and binary language models generated by the CMU-Cambridge Statistical Language Modeling Toolkit [22].

VI. Decoder

The primary role of the Sphinx-4 *Decoder* block is to use Features from the FrontEnd in conjunction with the SearchGraph from the Linguist to generate *Result* hypotheses. The Decoder block comprises a pluggable *SearchManager* and other supporting code that simplifies the decoding process for an application. As such, the most interesting component of the Decoder block is the SearchManager.

The Decoder merely tells the SearchManager to recognize a set of Feature frames. At each step of the process, the SearchManager creates a *Result* object that contains all the paths that have reached a final non-emitting state. To process the result, Sphinx-4 also provides utilities capable of producing a lattice and confidence scores from the Result. Unlike other systems, however, applications can modify the search space and the Result object in between steps, permitting the application to become a partner in the recognition process.

Like the Linguist, the SearchManager is not restricted to any particular implementation. For example, implementations of the SearchManager may perform search algorithms such as frame-synchronous Viterbi, A*, bi-directional, and so on.

Each SearchManager implementation uses a token passing algorithm as described by Young [24]. A Sphinx-4 token is an object that is associated with a SearchState and contains the overall acoustic and language scores of the path at a given point, a reference to the SearchState, a reference to an input Feature frame, and other relevant information. The SearchState reference allows the SearchManager to relate a token to its state output distribution, context-dependent phonetic unit, pronunciation, word, and grammar state. Every partial hypothesis terminates in an active token.

As illustrated in Figure 1, implementations of a SearchManager may construct a set of active tokens in the form of an *ActiveList* at each time step, though the use of an ActiveList is not required. As it is a common technique, however, Sphinx-4 provides a sub-framework to support SearchManagers composed of an *ActiveList*, a *Pruner* and a *Scorer*.

The SearchManager sub-framework generates ActiveLists from currently active tokens in the search trellis by pruning using a pluggable *Pruner*. Applications can configure the Sphinx-4 implementations of the Pruner to perform both relative and absolute beam pruning. The implementation of the Pruner is greatly simplifed by the garbage collector of the Java platform. With garbage collection, the Pruner can prune a complete path by merely removing the terminal token of the path from the ActiveList. The act of removing the terminal token identifies the token and any unshared tokens for that path as unused, allowing the garbage collector to reclaim the associated memory.

The SearchManager sub-framework also communicates with the *Scorer*, a pluggable state probability estimation module that provides state output density values on demand. When the SearchManager requests a score for a given state at a given time, the Scorer accesses the feature vector for that time and performs the mathematical operations to compute the score. In the case of parallel decoding using parallel acoustic models, the Scorer matches the acoustic model set to be used against the feature type.

The Scorer retains all information pertaining to the state output densities. Thus, the SearchManager need not know whether the scoring is done with continuous, semi-continuous or discrete HMMs. Furthermore, the probability density function of each HMM state is isolated in the same fashion. Any heuristic algorithms incorporated into the scoring procedure for speeding it up can also be performed locally within the scorer. In addition, the scorer can take advantage of multiple CPUs if they are available.

The current Sphinx-4 implementation provides pluggable implementations of SearchManagers that support frame synchronous Viterbi [18], Bushderby [25], and parallel decoding [26]:

- `SimpleBreadthFirstSearchManager`: performs a simple frame synchronous Viterbi search with pluggable Pruner that is called on each frame. The default Pruner manages both absolute and relative beams. This search manager produces Results that contains pointers to active paths at the last frame processed.
- `WordPruningBreadthSearchManager`: performs a frame synchronous Viterbi search with a pluggable Pruner that is called on each frame. Instead of managing a single ActiveList, it manages a *set* of ActiveLists, one for each of the state types defined by the Linguist. Pruning is performed in the decomposition and sequence order of the state types as defined by the Linguist.
- `BushderbySearchManager`: performs a generalized frame-synchronous breadth-first search using the Bushderby algorithm, performing classifications based on free energy as opposed to likelihoods.
- `ParallelSearchManager`: performs a frame synchronous Viterbi search on mulitple feature streams using a factored language HMM approach as opposed to the coupled HMM approach used by AVCSR [3]. An advantage of the factored search is that it can be much faster and far more compact than a full search over a compound HMM.

VII. Discussion

The modular framework of Sphinx-4 has permitted us to do some things very easily that have been traditionally difficult. For example, both the parallel and Bushderby SearchManager implementations were created in a relatively short period of time and did not require modification to the other components of the system.

The modular nature of Sphinx-4 also provides it with the ability to use modules whose implementations range from general to specific applications of an algorithm. For example, we were able to improve the runtime speed for the RM1 [27] regression test by almost 2 orders of magnitude merely by plugging in a new Linguist and leaving the rest of the system the same.

Test	*WER*		*RT*		
	Sphinx-3.3	*Sphinx-4*	*Sphinx-3.3*	*Sphinx-4 (1 CPU)*	*Sphinx-4 (2 CPU)*
TI46 (11 words)	1.217	0.168	0.14	0.03	0.02
TIDIGITS (11 words)	0.661	0.549	0.16	0.07	0.05
AN4 (79 words)	1.300	1.192	0.38	0.25	0.20
RM1 (1000 words)	2.746	2.739	0.50	0.50	0.40
WSJ5K (5000 words)	7.323	7.174	1.36	1.22	0.96
HUB-4 (64000 words)	18.845	18.878	3.06	4.40	3.80

TABLE I

SPHINX-4 PERFORMANCE. WORD ERROR RATE (WER) IS GIVEN IN PERCENT. REAL TIME (RT) SPEED IS THE RATIO OF UTTERANCE DURATION TO THE TIME TO DECODE THE UTTERANCE. FOR BOTH, A LOWER VALUE INDICATES BETTER PERFORMANCE. DATA GATHERED ON A DUAL CPU 1015MHZ ULTRASPARC®III WITH 2G RAM

Furthermore, the modularity of Sphinx-4 also allows it to support a wide variety of tasks. For example, the various SearchManager implementations allow Sphinx-4 to efficiently support tasks that range from small vocabulary tasks such as TI46[1] [28] and TIDIGITS[2] [29] to large vocabulary tasks such as HUB-4 [30]. As another example, the various Linguist implementations allow Sphinx-4 to support different tasks such as traditional CFG-based command-and-control applications in addition to applications that use stochastic language models.

The modular nature of Sphinx-4 was enabled primarily by the use of the Java programming language. In particular, the ability of the Java platform to load code at run time permits simple support for the pluggable framework, and the Java programming language construct of interfaces permits separation of the framework design from the implementation.

The Java platform also provides Sphinx-4 with a number of other advantages:

- Sphinx-4 can run on a variety of platforms without the need for recompilation
- The rich set of platform APIs greatly reduces coding time
- Built-in support for multithreading makes it simple to experiment with distributing decoding tasks across multiple threads
- Automatic garbage collection helps developers to concentrate on algorithm development instead of memory leaks

On the downside, the Java platform can have issues with memory footprint. Also related to memory, some speech engines will directly access the platform memory directly in order to optimize the memory throughput during decoding. Direct access to the platform memory model is not permitted with the Java programming language.

A common misconception people have regarding the Java programming language is that it is too slow. When developing Sphinx-4, we carefully instrumented the code to measure various aspects of the system, comparing the results to its predecessor, Sphinx-3.3. As part of this comparison, we tuned Sphinx-3.3 to get its optimal performance for both real-time speed (RT) and word error rate (WER). We then tuned Sphinx-4 to match or better the WER of Sphinx-4, comparing the resulting RT speeds. Table I provides a summary of this comparison, showing that Sphinx-4 performs well in comparison to Sphinx-3.3 (for both WER and RT, a lower number indicates better performance).

An interesting result of this comparison helps to demonstrate the strength of the pluggable and modular design of Sphinx-4. Sphinx-3.3 has been designed for more complex N-Gram language model tasks with larger vocabularies. As a result, Sphinx-3.3 does not perform well for "easier" tasks such as TI46 and TIDIGITS. Because Sphinx-4 is a pluggable and modular framework, we were able to plug in different implementations of the Linguist and SearchManager that were optimized for the particular tasks, allowing Sphinx-4 to perform much better. For example, note the dramatic difference in WER and RT performance numbers for the TI46 task.

Another interesting aspect of the performance study shows us that raw computing speed is not our biggest concern when it comes to RT performance. For the 2 CPU results in this table, we used a Scorer that equally divided the scoring task across the available CPUs. While the increase in speed is noticeable, it is not as dramatic as we expected. Further analysis helped us determine that only about 30 percent of the CPU time is spent doing the actual scoring of the acoustic model states. The remaining 70 percent is spent doing non-scoring activity, such as growing and pruning the ActiveList. Our empirical results also show that the Java platform's garbage collection mechanism only accounts for 2-3 percent of the overall CPU usage.

VIII. FUTURE WORK

Sphinx-4 currently provides just one implementation of the AcousticModel, which loads Sphinx-3.3 models created by the SphinxTrain acoustic model trainer. The SphinxTrain trainer produces HMMs with a fixed number of states, fixed topology, and fixed unit contexts. Furthermore, the parameter tying [5] between the SphinxTrain HMMs and their associated probability density functions is very coarse. Because the Sphinx-4 framework does not have these restrictions, it is capable of handling HMMs with an abitrary topology over an arbitrary number of states and variable length left and right unit contexts. In addition,

[1]TI46 refers to the NIST CD-ROM Version of the Texas Instruments-developed 46-Word Speaker-Dependent Isolated Word Speech Database.

[2]TIDIGITS refers to the NIST CD-ROM Version of the Texas Instruments-developed Studio Quality Speaker-Independent Connected-Digit Corpus.

the Sphinx-4 acoustic model design allows for very fine parameter tying. We predict that taking advantage of these capabilities will greatly increase both the speed and accuracy of the decoder.

We have created a design for a Sphinx-4 acoustic model trainer that can produce acoustic models with these desirable characteristics [31]. As with the Sphinx-4 framework, the Sphinx-4 acoustic model trainer has been designed to be a modular, pluggable system. Such an undertaking, however, represents a significant effort. As an interim step, another area for experimentation is to create FrontEnd and AcousticModel implementations that support the models generated by the HTK system [1].

We have also considered the architectural changes that would be needed to support segment-based recognition frameworks such as the MIT SUMMIT speech recognizer [32]. A cursory analysis indicates the modifications to the Sphinx-4 architecture would be minimal, and would provide a platform to do meaningful comparisons between segemental and fixed-frame-size systems.

Finally, the SearchManager provides fertile ground for implementing a variety of search approaches, including A*, fast-match, bi-directional, and multiple pass algorithms.

IX. Conclusion

After careful development of the Sphinx-4 framework, we created a number of differing implementations for each module in the framework. For example, the FrontEnd implementations support MFCC, PLP, and LPC feature extraction; the Linguist implementations support a variety of language models, including CFGs, FSTs, and N-Grams; and the Decoder supports a variety of SearchManager implementations, including traditional Viterbi, Bushderby, and parallel searches. Using the ConfigurationManager, the various implementations of the modules can be combined in various ways, supporting our claim that we have developed a flexible pluggable framework. Furthermore, the framework is performing well both in speed and accuracy when compared to its predecessors.

The Sphinx-4 framework is already proving itself as being “research ready,” easily supporting various work such as the parallel and Bushderby SearchManagers as well as a specialized Linguist that can apply “unigram smear” probabilities to lex trees. We view this as only the very beginning, however, and expect Sphinx-4 to support future areas of core speech recognition research.

Finally, the source code to Sphinx-4 is freely available under a BSD-style license. The license permits others to do academic and commercial research and to develop products without requiring any licensing fees. More information is available at `http://cmusphinx.sourceforge.net/sphinx4`.

Acknowledgments

The authors would like to thank Prof. Richard Stern at CMU, Robert Sproull at Sun Microsystems Laboratories, and Joe Marks at MERL for making this team possible. We also thank Sun Microsystems Laboratories and the current management for their continued support and collaborative research funds. Rita Singh was sponsored by the Space and Naval Warfare Systems Center, San Diego, under Grant No. N66001-99-1-8905. The content of this paper does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

References

[1] S. Young, “The HTK hidden Markov model toolkit: Design and philosophy,” Cambridge University Engineering Department, UK, Tech. Rep. CUED/F-INFENG/TR152, Sept. 1994.
[2] N. Deshmukh, A. Ganapathiraju, J. Hamaker, J. Picone, and M. Ordowski, “A public domain speech-to-text system,” in *Proceedings of the 6th European Conference on Speech Communication and Technology*, vol. 5, Budapest, Hungary, Sept. 1999, pp. 2127–2130.
[3] X. X. Li, Y. Zhao, X. Pi, L. H. Liang, and A. V. Nefian, “Audio-visual continuous speech recognition using a coupled hidden Markov model,” in *Proceedings of the 7th International Conference on Spoken Language Processing*, Denver, CO, Sept. 2002, pp. 213–216.
[4] K. F. Lee, H. W. Hon, and R. Reddy, “An overview of the SPHINX speech recognition system,” *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 38, no. 1, pp. 35–45, Jan. 1990.
[5] X. Huang, F. Alleva, H. W. Hon, M. Y. Hwang, and R. Rosenfeld, “The SPHINX-II speech recognition system: an overview,” *Computer Speech and Language*, vol. 7, no. 2, pp. 137–148, 1993.
[6] M. K. Ravishankar, “Efficient algorithms for speech recognition,” PhD Thesis (CMU Technical Report CS-96-143), Carnegie Mellon University, Pittsburgh, PA, 1996.
[7] P. Lamere, P. Kwok, W. Walker, E. Gouvea, R. Singh, B. Raj, and P. Wolf, “Design of the CMU Sphinx-4 decoder,” in *Proceedings of the 8th European Conference on Speech Communication and Technology*, Geneve, Switzerland, Sept. 2003, pp. 1181–1184.
[8] J. K. Baker, “The Dragon system - an overview,” in *IEEE Transactions on Acoustic, Speech and Signal Processing*, vol. 23, no. 1, Feb. 1975, pp. 24–29.
[9] B. T. Lowerre, “The Harpy speech recognition system,” Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, 1976.
[10] J. K. Baker, “Stochastic modeling for automatic speech understanding,” in *Speech Recognition*, R. Reddy, Ed. New York: Academic Press, 1975, pp. 521–542.
[11] P. Placeway, S. Chen, M. Eskenazi, U. Jain, V. Parikh, B. Raj, M. Ravishankar, R. Rosenfeld, K. Seymore, M. Siegler, R. Stern, and E. Thayer, “The 1996 HUB-4 Sphinx-3 system,” in *Proceedings of the DARPA Speech Recognition Workshop*. Chantilly, VA: DARPA, Feb. 1997. [Online]. Available: http://www.nist.gov/speech/publications/darpa97/pdf/placewa1.pdf
[12] M. Ravishankar, “Some results on search complexity vs accuracy,” in *Proceedings of the DARPA Speech Recognition Workshop*. Chantilly, VA: DARPA, Feb. 1997. [Online]. Available: http://www.nist.gov/speech/publications/darpa97/pdf/ravisha1.pdf
[13] F. Jelinek, *Statistical Methods for Speech Recognition*. Cambridge, MA: MIT Press, 1998.

[14] X. Huang, A. Acero, F. Alleva, M. Hwang, L. Jiang, and M. Mahajan, “From SPHINX-II to Whisper: Making speech recognition usable,” in *Automatic Speech and Speaker Recognition, Advanced Topics*, C. Lee, F. Soong, and K. Paliwal, Eds. Norwell, MA: Kluwer Academic Publishers, 1996.
[15] S. B. Davis and P. Mermelstein, “Comparison of parametric representations for monosyllable word recognition in continuously spoken sentences,” in *IEEE Transactions on Acoustic, Speech and Signal Processing*, vol. 28, no. 4, Aug. 1980.
[16] H. Hermansky, “Perceptual linear predictive (PLP) analysis of speech,” *Journal of the Acoustical Society of America*, vol. 87, no. 4, pp. 1738–1752, 1990.
[17] NIST. Speech recognition scoring package (score). [Online]. Available: http://www.nist.gov/speech/tools
[18] G. D. Forney, “The Viterbi algorithm,” *Proceedings of The IEEE*, vol. 61, no. 3, pp. 268–278, 1973.
[19] P. Kenny, R. Hollan, V. Gupta, M. Lenning, P. Mermelstein, and D. O’Shaugnessy, “A*-admissible heuristics of rapid lexical access,” *IEEE Transactions on Speech and Audio Processing*, vol. 1, no. 1, pp. 49–59, Jan. 1993.
[20] “Java speech API grammar format (JSGF).” [Online]. Available: http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/
[21] M. Mohri, “Finite-state transducers in language and speech processing,” *Computational Linguistics*, vol. 23, no. 2, pp. 269–311, 1997.
[22] P. Clarkson and R. Rosenfeld, “Statistical language modeling using the CMU-cambridge toolkit,” in *Proceedings of the 5th European Conference on Speech Communication and Technology*, Rhodes, Greece, Sept. 1997.
[23] Carnegie Mellon University. CMU pronouncing dictionary. [Online]. Available: http://www.speech.cs.cmu.edu/cgi-bin/cmudict
[24] S. J. Young, N. H. Russell, and J. H. S. Russell, “Token passing: A simple conceptual model for connected speech recognition systems,” Cambridge University Engineering Dept, UK, Tech. Rep. CUED/F-INFENG/TR38, 1989.
[25] R. Singh, M. Warmuth, B. Raj, and P. Lamere, “Classification with free energy at raised temperatures,” in *Proceedings of the 8th European Conference on Speech Communication and Technology*, Geneve, Switzerland, Sept. 2003, pp. 1773–1776.
[26] P. Kwok, “A technique for the integration of multiple parallel feature streams in the Sphinx-4 speech recognition system,” Master’s Thesis (Sun Labs TR-2003-0341), Harvard University, Cambridge, MA, June 2003.
[27] P. Price, W. M. Fisher, J. Bernstein, and D. S. Pallett, “The DARPA 1000-word resource management database for continuous speech recognition,” in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, vol. 1. IEEE, 1988, pp. 651–654.
[28] G. R. Doddington and T. B. Schalk, “Speech recognition: Turning theory to practice,” *IEEE Spectrum*, vol. 18, no. 9, pp. 26–32, Sept. 1981.
[29] R. G. Leonard and G. R. Doddington, “A database for speaker-independent digit recognition,” in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, vol. 3. IEEE, 1984, p. 42.11.
[30] J. Garofolo, E. Voorhees, C. Auzanne, V. Stanford, and B. Lund, “Design and preparation of the 1996 HUB-4 broadcast news benchmark test corpora,” in *Proceedings of the DARPA Speech Recognition Workshop*. Chantilly, Virginia: Morgan Kaufmann, Feb. 1997, pp. 15–21.
[31] (2003, Mar.) Sphinx-4 trainer design. [Online]. Available: http://www.speech.cs.cmu.edu/cgi-bin/cmusphinx/twiki/view/Sphinx4/Train%erDesign
[32] J. R. Glass, “A probablistic framework for segment-based speech recognition,” *Computer Speech and Language*, vol. 17, no. 2, pp. 137–152, Apr. 2003.