Novel LVCSR Decoder Based on Perfect Hash Automata and Tuple Structures – SPREAD –

Matej Rojc
Faculty of Electrical Engineering and Computer Science
University of Maribor
Maribor, Slovenia

Kačič Zdravko
Faculty of Electrical Engineering and Computer Science
University of Maribor
Maribor, Slovenia

***Abstract*— The paper presents the novel design of a one-pass large vocabulary continuous-speech recognition decoder engine, named SPREAD. The decoder is based on a time-synchronous beam-search approach, including statically expanded cross-word triphone contexts. An approach using efficient tuple structures is proposed for the construction of the complete search-network. The foremost benefits are the important space savings and higher processing speed, and the compact and reduced size of the tuple structure, especially when exploiting the structure of the key. In this way, the time needed to load the ASR search-network into the memory is also significantly reduced. Further, the paper proposes and presents the complete methodology for compiling general ASR knowledge sources into a tuple structures. Additionally, the beam search is enhanced with the novel implementation of a bigram language model Look-Ahead technique, by using tuple structures and a caching scheme. The SPREAD LVCSR decoder is based on a token-passing algorithm, capable of restricting its search-space by several types of token pruning. By using the presented language model Look-Ahead technique, it is possible to increase the number of tokens that can be pruned without decoding precision loss.**

Keywords—LVCSR decoder; tuple structure; finite automata; perfect hashing; Look-Ahead; language models

I. INTRODUCTION

A LVCSR decoder represents a major component in the development of any continuous speech-recognition system. Since tasks' and systems' complexities are constantly increasing, the decoder becomes an increasingly significant component within the overall development of compact and efficient speech-recognition systems. Therefore, more efficient designs can improve the trade-off between the needed decoding time and the recognition error rate. Furthermore, large knowledge sources are used in the LVCSR decoder, enabling estimation of the most likely word sequence from specific acoustic evidence. In general, these knowledge sources are: acoustic models (HMM - Hidden Markov Models), pronunciation lexicon, and N-gram language models. More and more new application areas require increase in the complexity of acoustic, lexicon and language models used in LVCSR decoders. Consequently, the requirements for time and space efficiency of LVCSR decoders are becoming greater and greater, despite the continuous growth of hardware performance, and GPU-like parallel processing. Therefore, efficient management of all these knowledge-sources, and efficient decoding of the acoustic input, still remains important issues and challenging tasks. Furthermore, in LVCSR decoders a lot of optimisation techniques, specific architectures, and heuristics have to be used and developed in order to achieve lower computational complexity and lower memory requirements. Progress regarding LVCSR decoding algorithms, together with the availability of ever increasing computing power and memory capacity, has also resulted in more accurate and close to real-time LVCSR decoders for tasks such as, e.g. broadcast news transcription, conversational telephone speech recognition systems etc. Technology based on weighted finite-state machines (WFSM) has already shown that it is possible to efficiently encode all those knowledge sources present within a speech-recognition system, such as e.g. language models, pronunciation dictionaries, context decision-trees, etc. By using them, a LVCSR network is usually obtained by a composition of several WFSTs. After using minimisation algoritms, an LVCSR network can be directly used in a Viterbi-based LVCSR decoder. These decoders have already been shown to yield good performance when compared to the classic approaches. This results in the implementations of several Viterbi-based decoders using FSA technology [7, 16, 17, 18, 19, 24]. Nevertheless, the complexity of acoustic and language models used in speech recognition tasks still imposes growing requirements for the efficiency and accuracy of LVCSR decoders, and fosters the development of new approaches and techniques such as, e.g. cross-word acoustic models and long-span language models, already resulted in the development of several solutions for the speech-decoding problem [1, 2, 5, 6, 8, 10, 21, 22].

II. RELATED WORK

Nowadays, a lot of speech-decoding software packages exist that employ a number of different decoding techniques, based on time synchronous Viterbi search and many are also available for research purposes. CMU/Sphinx, released by Carnegie Mellon University (CMU) [26], contains less features and flexibility, but in contrast to HTK [27], focuses more on speed and was one of the first ASR systems to offer support for speaker-independent Large Vocabulary Continuous Speech Recognition (LVCSR) system. Latest versions, although less efficient than previously, are more flexible and enables faster and easier development and maintainance of different applications [17]. Further, the Julius LVCSR decoder is a high-performance, two-pass decoder, focusing on performance, modularity, and availability [9]. The HTK framework is very flexible and comprises a lot of state-of-the-art ASR features,

e.g. vocal tract length normalization (VTLN), heteroscedastic linear discriminant analysis (HLDA) and discriminative training with maximum mutual information (MMI), and minimum phone error (MPE) criteria [15,27]. Some of the decoder implementations have shifted from dynamic search to static graphs in the form of probabilistic weighted finite-state transducers (WFSTs) [4, 7, 18]. Architectures based on the theory of weighted finite-state transducers represent flexible and efficient decoder architectures. The advantages of this implementation can be seen in the simplicity of LVCSR decoders and the seamless composition of lexicon, acoustic, and language models. One of the most efficient solutions for search-network optimization is the WFST framework from [12]. In these architectures, all the knowledge sources are combined together statically. Furthermore, the search network can be optimised for maximal efficiency. In such LVCSR systems the decoding network is usually compiled independently of the LVCSR decoder itself, in this way representing also more flexible solutions for e.g. the incorporation of several application-specific knowledge sources. Nevertheless, Mohri's approach can restrict the complexity of the knowledge sources, and prevent some on-the-fly adaptation [22]. A drawback can also be seen in the memory requirements for the compilation of very large static decoding networks for LVCSR systems, although today this issue has become less crucial because of the availability of 64-bit systems, and a lot of available RAM memory. Another approach from [25] expands the search network dynamically. This approach, on the other hand, can be computationally too expensive for efficient decoding regarding larger LVCSR tasks. Recently, the Juicer WFST decoder has become a popular WFST-based alternative to the tree-based dynamic decoders, as provided with the HTK and Sphinx toolkits. The T3 WFST decoder is a system that performs favorably against several established decoders in the field, including the Juicer, Sphinx, and HDecode [27] in terms of RTF versus Word Accuracy [17]. In the case of the T3 WFST decoder, in addition to the existing HTK conversion tool, a tool has also been developed for converting arbitrary Sphinx format acoustic models into a format suitable for use with the T3 WFST decoder [4]. Juicer provides similar functionality to the T3 WFST decoder in terms of the model inputs it accepts. It is capable of performing decoding on both static cascades, as well as on-the-fly composition, and it has been developed to read in HTK-based acoustic models in native format [13].

This paper proposes a novel LVCSR decoder named SPREAD that is implemented by using efficient perfect-hash automata and tuple structures. The complete search network is compiled independently of the LVCSR decoder (off-line), including the needed pronunciation lexicon, language models and Look-Ahead structures, and can be fast loaded by the runtime system. The language model information is, in this way, dynamically obtained during the search within the LVCSR decoding engine. Furthermore, the proposed off-line compilation methodology of large static networks is simple and fast, even on 32-bit machines with less available RAM memory. In this LVCSR decoder, the novel implementation of the language model Look-Ahead technique (used to enhance the beam search results), based on tuple structures, is further integrated. In this way, the proposed LVCSR decoder incorporates several novel design strategies, which have not been used earlier in conventional decoders of HMM-based large vocabulary speech recognition systems. The paper starts with a motivation for using tuple structures in speech-technology-related applications, and for developing a LVCSR decoder, based on tuple structures. Firstly, the formalism behind tuple structures is presented, regarding their form and representation. Then the presentation of the LVCSR decoder technology used within the LVCSR SPREAD decoder follows. Next, the proprietary FSM tools are discussed and their application for the construction of tuple structures when developing LVCSR decoders. The main part of the paper represents the proposed compilation methodology of all the tuple structures used within a general LVCSR decoder. Additionally, the implementation of a large-scale search-network using tuple structures is described in detail. The proposed work is based on real implementation of the LVCSR decoder based on tuple structures, as used for 64k broadcast news transcription speech-recognition task for the Slovenian language. Statistics and achieved compactness of the proposed implementation are, therefore, presented in Section 8. In this way, the paper familiarizes the readers with the design solutions encountered, when building a tuple-based LVCSR decoders. The conclusion is drawn at the end.

III. MOTIVATION

The general practical issue in LVCSR speech-recognition applications concerns the size of the knowledge sources, and the size of the complete search-network. This issue can be even more crucial when the knowledge sources are consulted frequently and must, therefore, be loaded into the memory. Perfect hashing techniques based on finite-state automata can be very efficient when solving these problems [3, 23]. Namely, as will be shown, they enable compact representations without sacrificing the lookup time. In the case of LVCSR speech-recognition applications, large dictionaries are not the only space-consuming resources. Namely, several types of language models containing statistical information about the co-occurrence of words, require even more memory space, and also at the same time as fast lookup operations as possible – LVCSR systems need to be capable of working with e.g. bigram, trigram, fourgram models etc. Therefore, for speech-recognition applications, the achievable size and compactness of language models and other knowledge sources within the runtime system represent an important practical implementation issue, and also motivation for the work presented in this paper. The Slovenian language is a highly inflectional language. Therefore, the number of distinct word forms in everyday use is very large, resulting in large knowledge sources for general LVCSR speech recognition tasks. When considering this, efficient management of the data structures' size when representing knowledge sources, and the lookup efficiency, are general requirements. In this respect a very compact representation of knowledge sources and the search-network is needed, and a highly-optimized LVCSR ASR decoder must be implemented. In order to better cope with this problem, it was decided to work on a new design approach for the development of an LVCSR decoder that is based completely on perfect hash automata and the so-called tuple structures, by following the established theory on tuples

in [3]. In general, the needed knowledge sources for LVCSR ASR decoder can be represented in the form of a simple data structure that defines a mapping from some strings to some value. These data structures can be easily generalized, in which the keys are *n*-tuples of strings, for a fixed *n*. Such data structure is called tuple structure, and apart from the N-gram language models, it can also be used for the creation of large ASR search-networks including LMLA (language model look-ahead) info, as presented in the following sections. Although operations on tuple structures, like insertion and deletion, are not well supported, they can be ignored in the case of LVCSR decoders, since tuple structures can be constructed once from a given data-set (off-line), and then only loaded and used within the runtime system. At the end, also compact representation of the tuple structure is very important. Namely, by compact representation of knowledge sources using tuple structures, the time needed to load the structures into the memory is significantly reduced. The techniques used in the presented work, can be seen as applications and extensions of perfect hashing based on finite-state automata. Therefore, the proposed implementation yields to flexible and compact representation of large scale knowledge sources and also LVCSR search-networks in practice. The following section presents the basic formalism behind tuple structures.

IV. Tuple Structures

A tuple structure $T^{i,j}$ is a finite function $(W_1 \times ... \times W_i) \rightarrow Z^j$. In this finite function, $W_1 \times ... \times W_i$ are simple sets of strings, and Z are the integers [3]. This finite function can map to a tuple of integers, or to a tuple of real numbers. The so-called *word columns* contain words (e.g. lexicon words, pronunciations, diphones, triphones, language models' pairs, Look-Ahead pairs etc.). And the so-called *number columns* contain one or several integer numbers, or real numbers (e.g. N-gram probabilities, N-gram backoff-weights). Figure 1 presents part of the table forms used for the construction of tuple structures for a bigram language model, and for one of the layers within the LVCSR search network. In the first case, the word columns contain word sequences, and the number columns the bigram probabilities and backoff-weights. In the second case, the word columns contain triphones and diphones, and the number columns contain the next layer ID and the next node type (context or model node). Perfect hash finite automata are needed for the tuple structures. The perfect hash finite automaton for a finite set of words *W* is such minimal deterministic acyclic finite automaton *N* that accepts each word in *W*. And each transition within the automaton has an assigned integer number *j*. Let some word *w* represents the *i*-th word of *W*. Then the sum of the integers along an accepting path in *N* is *i*. If *N(w)* refers to the *hash key* assigned to *w* by *N*, then the time spent for its computation is $O(|w|)$ [3]. The perfect hash automata are needed in order to represent all the words in the word columns with hash keys. Furthermore, they can be used within the LVCSR decoder, when translation from the hash keys back into words is needed (e.g. ASR output results etc.). When there is enough overlap between the words from several word columns within the table forms, the same perfect hash automaton for all those columns can be used. Although, the tuple structures are able to take advantage of such shared dictionaries, it is not required that the dictionaries for different word columns are the same.

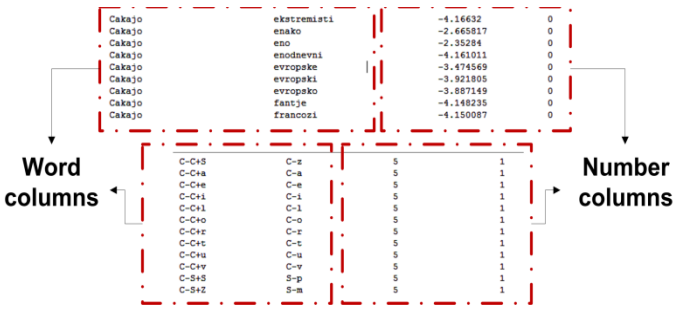

Fig. 1. Table forms consisting of word and number columns.

In general, several hash automata are used (one for each word column). Nevertheless, in the first case more space savings can be achieved. Figure 2 then shows the representation of word columns by the corresponding hash keys for the bigram language model and for one of the layers used in the ASR search-network.

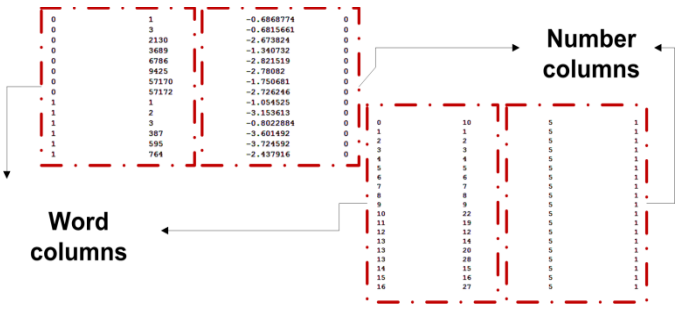

Fig. 2. Representation of word columns in table forms by using the hash keys.

A. A Table Form for Compact Representation of Tuple structures

The tuple structure $T^{i,j} : (W_1 \times ... \times W_i) \rightarrow Z^j$ is in general represented by maximal *i* perfect hash automata (when each word column has its own perfect hash automaton). Then, for each tuple structure a table form consisting of *i*+*j* rows is constructed (Figure 2). The table forms are constructed for each sequence $w_1 ... w_i$ in the domain of *T*. For *T* we have the following transformation $T(w_1 ... w_i) = (z_1 ... z_j)$. The sequences of words $w_1 ... w_n$ are converted into their hash-keys $N(w_1) .. N(w_n)$ by using perfect hash automata. In this way, each word sequence is represented by a row in the table, consisting of $N(w_1), ..., N(w_i), z_1, ..., z_j$ [3]. As can be seen in Figure 2, all the cells in the table contain numbers at the end. For compact representation it is, therefore, important that each hash-key is represented with as few bytes as are required by the largest number within individual column. An additional benefit is the machine-independency of such representation. The tables also have to be sorted in order to guarantee sorted and unique entries. At the end, the tuple structure is represented by a table of packed numbers and *i* perfect hash automata that can be used for translating words into corresponding hash-keys and vice

versa. In order to access value(s) for a given sequence, a query string is needed, consisting of hash-keys. A binary search is used to find the corresponding entry within the table. The data for a given sequence can be obtained after an unpacking of the values found in the table is performed. The time needed for calculating the hash-keys is proportional to the combined length of words within the table's entry. The binary search takes $O\left(\log\left|T^{i,j}\right|\right)$ time and is proportional to the logarithm of the number of tuples. Tuple structures $T^{i,j}$ can also be constructed when i=1 (there is only one word column, e.g. unigrams for language model). In this case, the words in the word column are unique, therefore, their hash keys are also unique numbers from $0...|W_1|-1$. Consequently, there is no need to store the hash keys of the words within the table. Instead, the hash-keys just serve as an index within the table. Also the lookup function is different. After the hash-key is obtained, it is used as the address of the numerical tuple.

B. Tree representation

The hash-key in the first column of the table can be the same for many rows (e.g. in Figure 2). On the other hand, a particular instance of initial words $w_1...w_k, k<n$ within a tuple may appear several times. The so-called *trie* structure is obtained when representing them only once, and providing a pointer towards the remaining part, and performing the same steps recursively for all the remaining columns. The corresponding edges from the root are labelled with all the hash-keys used in the first column. These edges then point towards the following vertices with outgoing edges, thus representing tuples that have the same two words at the beginning etc. In order to economize the storage space, only one copy of the hash-keys from the first few columns is kept. Additional memory for the pointers is also needed. Each vertex is represented as a vector of edges. Each edge then consists of the label (hash-key) and a pointer that always points to the first son of the vertex. In this way, the number of sons for a specific vertex can be defined as the difference between the pointer for the current vertex, and the pointer for the next one. Such representation works best if the table is dense, and if it has very few columns.

According to [3], it is necessary to only construct the trie from the word columns. Namely, the numerical columns are the corresponding output, and can be kept intact. Furthermore, the overall size of the trie structure must be minimal. Therefore, the sizes of the used pointers should be as small as possible. Each level of the trie structure corresponds to a word column of a table, and is kept separate from other word columns. Next, each word column has a separate address. Pointers only point to the next column. In this way, they represent an index within the next column (is the ordinal number of the entry within the column that they point to), and not an index in all nodes of the trie. At all trie levels (except for the last one) all vertices have at least one son. Therefore, it is possible to store a given pointer again as a difference between the index of the item it points to, and the index of the current (pointing) item. The difference will always be non-negative. The size of the pointer is defined as the smallest number of bytes needed to represent the difference between the number of items within the next column and the number of items within the current one. We don't need pointers for the last column. Namely, its indexes are the same as those in the numerical part of the tuple. Let's e.g. try to access entry $T(w_1...w_i)$. First, the value $N(w_1)$ is calculated, and then follows the search for it within the first column. When the value is not found, then the searched entry $T(w_1...w_i)$ is not stored in the tuple. On the other hand, if the value $N(w_1)$ is found, the next value $N(w_2)$ has to be calculated and then searched in the specific portion of the second column. This portion is defined by the pointer found by $N(w_1)$. The portion end is defined by the pointer at the next hash-key value in the first column. The process continues into the next columns in the same way until reaching the hash key of the last word (or fail). The index of the hash-key for the word in the last column also represents the index in the numerical part of the tuple. Binary search is best to use to find the appropriate keys in specific portions of the word columns.

A special case are those tuple structures $T^{i,j}$, where *i=1*. In this case there is no need to store the hash-keys of the words in the first word column. As the first word column is also the last one, there are also no pointers. Each hash-key of the first word column is just an index to the numerical part of the tuples.

C. Representation of real numbers

Especially in the case of N-gram language models, the number columns containing the N-gram probabilities and the backoff-weights, that demand most space. Therefore, their compact representation is even more important. Further, different computer platforms represent real numbers in a different way, using various precision. Therefore, porting numbers from one computer to another many times also results in loss of precision. The precision of a representation can be increased when we use more bytes. But in this case, the goal is also to achieve as compact a representation as possible (in the case of real numbers). Knowledge sources for the LVCSR decoder, in general, contain N-gram language models with real numbers that are frequently represented in textual form (e.g. ARPA language model). Obviously, loss of precision in this case has already happened and the precision of the representation as used in the LVCSR decoder cannot be any higher. When considering this, it is possible to specify the precision of the real numbers within the tuple data, based on the number of digits in the mantissa. In the case of ARPA language models, it is assumed that only the digits presented in the textual form of a number are significant. Then, each real number in a specific number column is decomposed into a normalized mantissa *m* and an exponent *t*, such that $r = m \cdot 2^t$, $|m| \le 0.5$ or $m = 0.0$. Let $\hat{m} = a_0.a_1...a_n$ be a representation of a mantissa *m*, where $\hat{m} = \sum_{i=0}^{n} a_i \cdot 2^{-i}$. The precision ε is then 2^{-n-1} (the biggest number, where $|\hat{m} - m| < \varepsilon$). In this way, at least $\lceil n/8 \rceil + 1$ bytes are needed to represent the mantissa with precision ε. The number of needed bytes for the exponent is also calculated, but in all practical applications it is one [3]. In the following sections, the SPREAD LVCSR decoder is presented in detail, especially by describing the implementation

of knowledge sources and the ASR search-network when using tuples.

V. LVCSR DECODER – SPREAD –

The SPREAD LVCSR decoder has been designed with a high degree of modularity. Figure 3 presents the high-level architecture of the decoder. The decoder architecture consists of four main blocks that are defined, or controlled depending on the specific application in mind. The code within each module is modularly and flexibly structured, thus enabling flexible configurations of the decoder engine.

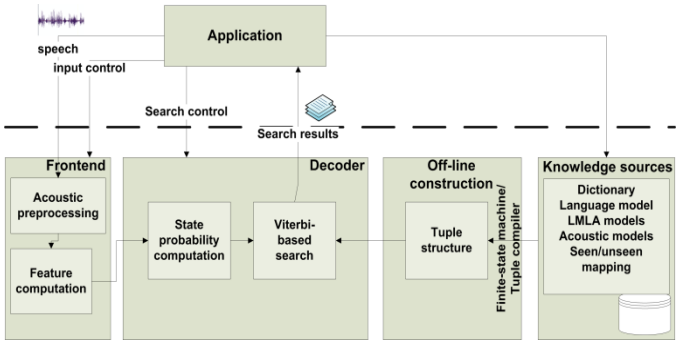

Fig. 3. Architecture of the SPREAD LVCSR decoder, based on the tuple structures.

By using tuple structures, the language-dependent knowledge sources are separated from the decoder. Furthermore, the proposed methodology of constructing one compact tuple structure (large ASR search-network, with an N-gram language model and LMLA information included), is performed off-line. Within the runtime decoder, the tuple structure is then loaded and used within the Viterbi-based search engine. Since the loading of the compact tuple structure is fast, even for large knowledge sources, the decoder is able to switch between several knowledge sources quickly and efficiently - even within a runtime system. All LVCSR decoder modules are written in C++ programming language. The off-line methodology for constructing a tuple structure from knowledge sources is performed by set of Perl[1] scripts, using several proprietary C++ FSM tools, as presented in the next sections.

Specific application defines the knowledge sources that, in general, consist of lexical, phonetic, and acoustic knowledge. The *lexical knowledge* consists of known words, along with their corresponding pronunciations. Additionally, multiple pronunciations can be included with a prior probability for each pronunciation variant. The *phonetic knowledge* consists of fundamental units within the pronunciation lexicons that are modelled in the context of their neighbours. In this way they account for the systematic and contextual variations that can be found in natural spoken speech across word boundaries. The *acoustic knowledge* is described by way of the state emission probability density functions (PDF) associated with each state of each context-dependent phoneme. Several parameters tying schemes can be used in estimation of emission PDF. The frontend module takes care for acoustic pre-processing, and the parameterisation of the speech data. The SPREAD LVCSR decoder-block then performs the recognition. The decoding problem within the system is to find the most likely word sequence $W_1^n = w_1, w_2, ..., w_n$, given a sequence of acoustic observation vectors $O_1^T = o_1, o_2, ..., o_T$, obtained from the speech signal. According to the theory in [14], this can be described by the following equation:

$$\hat{W} = \arg\max_{W_1^n} \left\{ P(W_1^n) \cdot P(O_1^T | W_1^n) \right\} = \arg\max_{W_1^n} \left\{ P(W_1^n) \cdot \sum_{S_1^T} P(O_1^T, S_1^T | W_1^n) \right\} \quad (1)$$

where W_1^n stands for the sequence of words, $S_1^T = s_1, s_2, ..., w_T$ represents any state sequence of length *T*, and $P(W_1^n)$ comprises the language model (LM) representing prior linguistic knowledge independently of the observed acoustic information. In the SPREAD LVCSR decoder, this is carried out by using a stochastic N-gram, where word probabilities are only dependent on the N-1 predecessor, and $P(O_1^T | W_1^n)$ represents the model of the lexical, phonetic, and acoustic knowledge. A complete search through such a space is still practically infeasible. Therefore, a number of approaches exist that try to solve this decoding problem. In the SPREAD LVCSR decoder, a time-synchronous search approximates the solution of the previous equation, by searching only for the most probable state sequence:

$$\hat{W} \cong \arg\max_{W_1^n} \left\{ P(W_1^n) \cdot \max_{S_1^T} P(O_1^T, S_1^T | W_1^n) \right\} \quad (2)$$

Decoding within the SPREAD LVCSR decoder performs a time-synchronous search of a network of hypotheses. At each time-step only the best hypotheses arriving at each state are retained and, in order to improve the efficiency, only the most likely hypotheses are extended to the next time-step. As already mentioned, the decoder block does not construct the ASR search-network within the runtime system. Namely, it is constructed off-line in the form of one common tuple structure that is loaded into the system during initialisation, or changed any time during the on-line process. The final tuple structure combines a standard N-gram language model, pronunciation dictionary, Look-Ahead information, and seen/unseen triphones mapping info. The decoder block is based on the token-passing algorithm with beam-search, and histogram pruning. At run time, the decoder expands the model-level tuple structure-based network into a state-level network that is suitable for finding the best state-level path. The search module requires likelihood scores for any current feature vector, in order to generate the active list. The likelihoods are computed by the state probability computation module that has access to the feature vectors.

VI. FSM TOOLS FOR THE CONSTRUCTION OF TUPLE STRUCTURES

An important advantage when using tuples for speech decoding is that they enable the integration and optimisation of several knowledge sources under the same generic representation. The proposed methodology for compiling knowledge sources into common tuple structure is performed by using proprietarily developed FSM tools, based on the theory and tuple technology as proposed in [3]. In Figure 4, the *fsmbuild*, *fsmhash*, and *fsmtuple* are those tools needed for

[1] http://www.perl.org/

compiling ASR knowledge sources into a corresponding tuple structures. Each ASR knowledge source can be split into N word columns, and M data columns. Further, the input data has to be sorted. Then perfect hash automata are built for word columns (by using the *fsmbuild* tool). In this way, a finite-state automaton is obtained that recognizes all words within individual word column (representing e.g. the triphones, diphones etc.) of the given knowledge source.

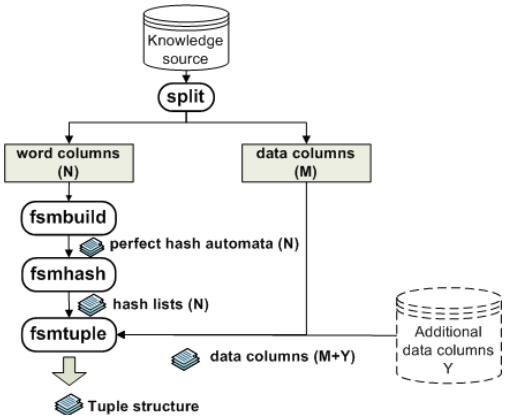

Fig. 4. FSM tools used for compiling ASR knowledge sources.

The perfect hash automata provide the mapping between words and a range of integer numbers – hash-keys. The exact numbering of the words is important for the tuple construction process. The perfect hash-automaton is at the end written into the file in binary form: a table of structures corresponding to arcs, with each arc containing a label, the number of arcs that lead from the node the arcs point to, and the index of the first arc that leads from the node the given arc points to. The *fsmhash* tool is used for translating words in specific word columns of the given knowledge source (e.g., diphones, triphones, words, etc.) into unique hash-keys. The input to the tool represents N built perfect hash automata and the corresponding N word columns' lists created beforehand. The outputs are N hash lists. In any step within the SPREAD LVCSR decoder, the mappings from hash numbers back into strings, and vice versa, can be easily and efficiently performed using these perfect hash automata. The N hash lists and additional data columns (containing integer or real numbers) are stored as table forms. At the end, the *fsmtuple* tool creates a compact structure, named the tuple structure. As can be seen from Figure 5, the input for the tuple construction process is represented in table form (*.llist), consisting of *N* columns representing words (as hash-keys), and several numbers' (integer or real) columns *M+Y*, representing tuple's data. The number of all columns *n* has to be specified, and the number of word columns *w* in a table. Furthermore, the size of the mantissa *s* can be specified (or calculated). The hash-keys for words have already been computed before, using the *fsmhash* tool. Therefore, the first step is to find the sizes of these hash-keys for each word column. Then the size of numbers in the numerical part is determined. The numerical part can contain integer or real numbers. The mantissa and exponent are calculated in the case of real numbers. The size of the whole number is, in this case, the sum of the mantissa size and the exponent size. In the case of integer numbers, the size of the numbers is just calculated. All these sizes are calculated as the number of needed bytes for storing the numbers within a specific column. In this way, only so many bytes as needed are used, to correctly represent any float or integer number within the columns of the table. Next, the tuple is constructed from the input table and written into the file. All data are written as bytes. Therefore, dedicated functions for converting numbers into bytes are used. Their input arguments are corresponding number and the calculated number of bytes that has to be used for its representation in bytes. As shown in Figure 6, the header is first written into tuple, containing e.g. the version, the word/number structure as described in the table, etc. Then, the sizes of the numbers for each column and sign vector (columns can also contain negative numbers) are written. Next, the calculated mantissa size is written for each column in the numerical part.

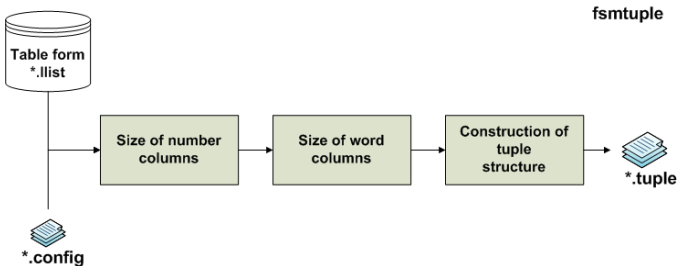

Fig. 5. The steps used during the tuple-structure construction process.

All data stored in the tuple structure is needed in order to correctly restore any number from bytes. Then, follows the construction of the tree structure: creation of the root node, with a list of pointers (1 for each child). These pointers point to records (ordered lists) of several fields e.g. hash-key, reference to a subtree etc. At the end, the indexes for the whole tree are calculated. Namely, the individual nodes of the tree are accessible via pointers from their parents. Nevertheless, in the tuple the pointers are replaced with indexes, being the ordinal numbers of the nodes within the corresponding layers. Then, the size of the whole tree is calculated (in bytes) and written into the file. Based on the indexes calculated and stored before, it is now possible to calculate and store addresses (at the byte level) of all columns in the tuple. This step is only needed when there is more than one word column in the table. At the end, the tree is also written into the tuple file (tree node's IDs, corresponding numerical part as data etc.). In this way, the tree and the corresponding numerical data are represented in the form of bytes. Such a structure is also easily stored as a binary file. In *fsmtuple* tool's configuration file, only the number of all columns has to be defined, and a number of word columns within the table. Additionally, the developer can optionally specify the size of the mantissa (can also be calculated), the desired separator between the columns in the table, the desired precision for the real numbers, the tuple's output file name, etc. All FSM tools are written in C++ programming language.

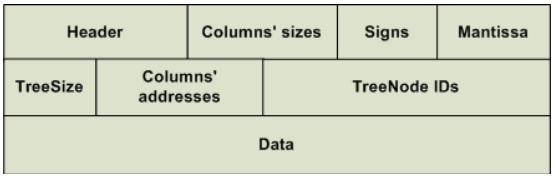

Fig. 6. Binary representation of the tuple structure.

VII. Application of Tuples To The Spread Lvcsr Decoder

In this section we propose a novel design for a one-pass LVCSR decoder engine SPREAD, based on tuple structures. The application's specific knowledge sources and ASR search-network are represented in the form of tuple structures that are combined within compact tuple-based decoding network. All these steps can be performed off-line. The detailed architecture of the LVCSR decoder SPREAD is presented in Figure 7.

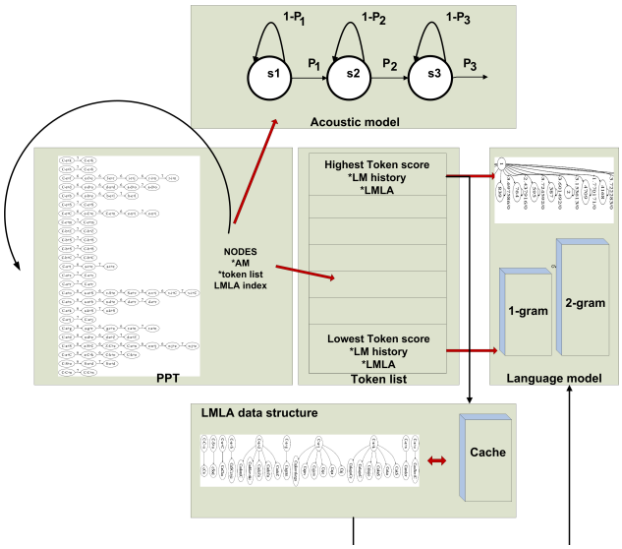

Fig. 7. The architecture of the tuple-based LVCSR decoder SPREAD.

The Viterbi search of the decoder is implemented using the token passing paradigm [27]. Hidden Markov Models (HMM) applying Gaussian Mixture Models (GMM) and the n-gram back-off language models are used to calculate the acoustic likelihoods of the context-dependent phones, and to calculate the language probabilities, respectively. The HMMs are organized within a static pronunciation prefix tree (PPT), as described in [8]. Each token contains a pointer to its LM history. Tokens coming from the leaves of the PPT are fed back into the root node of the tree after their *n*-gram history is updated. Token collisions will only occur for tokens with the same LM history. This means that each HMM state of each node in the PPT can contain a list of tokens with unique *n*-gram histories. These lists are sorted in descending order of the token probability scores.

Furthermore, decoders that make use of token-passing, restrict their search-space by various types of token pruning. In PPT-based decoders the global pruning and word-end pruning are commonly used [8]. Within the LVCSR decoder SPREAD both beam pruning methods are supported. In the case of beam pruning, tokens with a probability value between the best found probability and the best probability minus a constant beam are retained at each time-frame. All those tokens that do not fall within this beam are deleted.

During global beam pruning all tokens of the entire PPT are also compared to the best scoring token, and pruned if necessary. Word-end beam pruning is performed on all tokens that are at the leaves of the PPT, and for which the LM probabilities are incorporated into their probability scores. This pruning method is used to limit the number of tokens that are fed back into the root node of the PPT. Histogram pruning can also be used in the LVCSR decoder. Here, only the best N tokens are retained, when the number of tokens exceeds the maximum N (we significantly restrict required memory). Similar to beam pruning, histogram pruning can be performed both globally (global histogram pruning), and also in the leaves of the tree (word-end histogram pruning).

By using the proposed language model look-ahead (LMLA) technique based on tuple structures, it is possible to increase the number of tokens that can be pruned without any loss of decoding precision. It is well-known that, in the case of token-passing decoders that use PPT, full *n*-gram LMLA considerably increases the needed number of language model probability calculations. The SPREAD LVCSR decoder uses a full n-gram LMLA with a single static PPT, which is based on the tuple structures and efficient caching mechanism. Additionally, an LMLA index is assigned to each PPT's node, and an index to an LMLA field is added to each token list. The N-gram language model is also implemented in the form of a tuple structure. The language model knowledge is added to the hypothesis score at the PPT leaf nodes, and used by the LMLA mechanism.

In the following subsections the proposed methodology for constructing a compact ASR search-network based on tuple structures for a LVCSR decoder SPREAD, is presented in detail.

A. Compiling N-gram language models

Compiling N-gram language models (LM) into a tuple structure is also performed off-line. In the presented LVCSR decoder configuration, the input represents the LM N-gram language model stored in ARPA format, as shown in Figure 8. The separation into 1-gram and 2-gram data is performed first. Each file consists of word and number columns, representing unigram/bigram probabilities, and backoff weights. Next, each file is split into separate word columns and number columns, since different tasks have to be performed on each of them. Pre-processing has to be performed, in order to obtain unique and sorted lists for each word column.

The sorted word lists are then fed into the *fsmbuild* tool, and the corresponding perfect hash automata are built. In the next step the *fsmhash* tool is used, in order to translate all the words in the word columns into the corresponding lists, using hash-keys. Namely, for a final LM table form, hash keys are needed instead of words.

Furthermore, by using perfect hash automata, it is possible to translate hash-keys back into words effectively and efficiently, and vice versa, when needed. The obtained hash lists and number (data) columns are then merged into the table, by specifying the desired separator between the columns, and given to the *fsmtuple* tool. Its output then represents a LM tuple structure that has an efficient and compact trie structure. Basically, two separate tuple structures are built, and then merged into one. One structure is constructed for unigrams, and the other for bigrams.

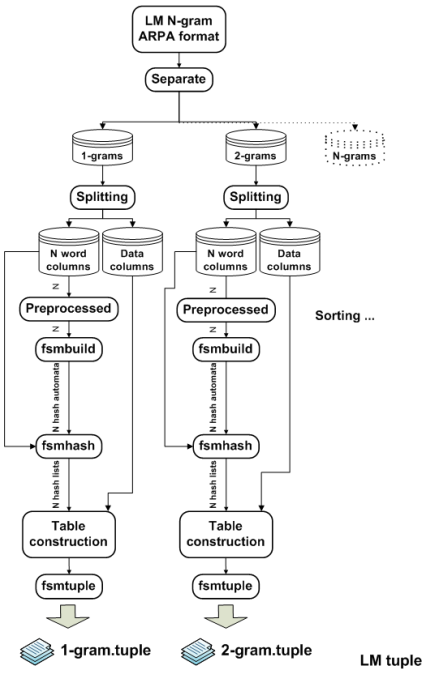

Fig. 8. Compilation process for N-gram language models.

B. Compiling the LVCSR search network

The main step within the proposed methodology for compiling ASR knowledge sources into a tuple structure represents the tuple-based construction of the ASR search-network. This structure can be constructed off-line, and is based on the idea of static PPT, and the work done in [27]. The traditional phone-level tree can be made more efficient by utilizing HMM level state tying, which has also been implemented. Cross-word triphone contexts are handled by compiling several tuple structures, with which the PPT tuple structure is merged at the end of the procedure. The obtained tuple-based network structure is in this way very compact. A general search network consists of nodes that are linked to each other with arcs.

These nodes can either correspond to one HMM state, or be dummy nodes without any acoustic probabilities associated with them. During decoding, the dummy nodes are passed immediately. They only mediate the tokens used to present the active search-network. A node can also have a word identity associated with it, which leads to the insertion of the word into the word history of the token passing that node. The proposed procedure of compiling such a LVCSR search-network into a tuple structure, assumes triphone models, where every triphone is defined in the acoustic models, and they are not tied at the triphone level. Instead, each triphone has a set of HMM states (three states in a left-to-right topology), and these states are shared amongst all triphones.

The state tying is performed using a decision tree. In this way the SPREAD LVCSR decoder is based on tuple-based network topology, including cross-word triphone models. The proposed methodology of compiling a search-network into a tuple structure follows the classical network topology idea, which is described with nodes and transition links, where the nodes are ordered in several layers [27].

Such a network also uses application specific vocabulary, and a HMM model set. The goal was to build a compact tuple representation of such a network topology, and integrate within it all the needed knowledge sources, like tuple-based N-gram language models, and tuple-based LMLA info. Construction of the LVCSR search-network to be used by the SPREAD LVCSR decoder is performed off-line, and can be repeated for any other ASR knowledge source available for application. The proposed methodology is presented in Figure 9.

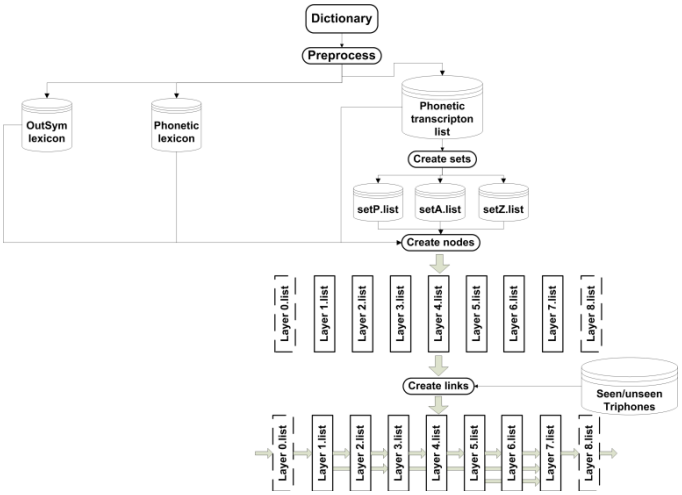

Fig. 9. The construction of the tuple-based LVCSR search network (first step).

The input represents a large dictionary. Firstly, the *outsym* and *phonetic* lexicons are built. A phonetic lexicon can be viewed as a list of word entries, where each entry contains orthography for the word and a corresponding list of pronunciations. A phonetic pronunciation in the dictionary can also contain a so-called output symbol. It is optional, but when present, the recognition output can use the specified output symbol rather than the word itself.

Therefore, an additional *outsym* lexicon can be built when this info is available. Additionally, a phonetic transcription list is built, containing only phonetic transcriptions for all the words. This list is then used for the construction of three phone sets, named *P* (all 1- phoneme transcriptions), *A* (first phonemes of all transcriptions), and ***Z*** (last phoneme of all transcriptions). All these sets are used in the next step for the construction of nodes within all layers of the network topology, here numbered from 0 to 8.

Layers 0, 2, 4, and 6 are those layers with model nodes, the other layers are used for context nodes. This step includes creation of the word final nodes (layer 0), silence (*sil/sp*) nodes, sentence start node (layer 3) and sentence end node (layer 5), word initial nodes (layer 4), PPT nodes (layer 6), word end nodes (layer 7), and other context nodes in layers 1, 3, 5, and 8. *1*-phoneme words are represented with corresponding nodes in layer 3. The model nodes are actually triphones, and other nodes are diphones. All the model nodes are firstly represented by linguistic triphones (using linguistic phonetic transcriptions from the dictionary), and then replaced by acoustic ones using seen/unseen mapping lists.

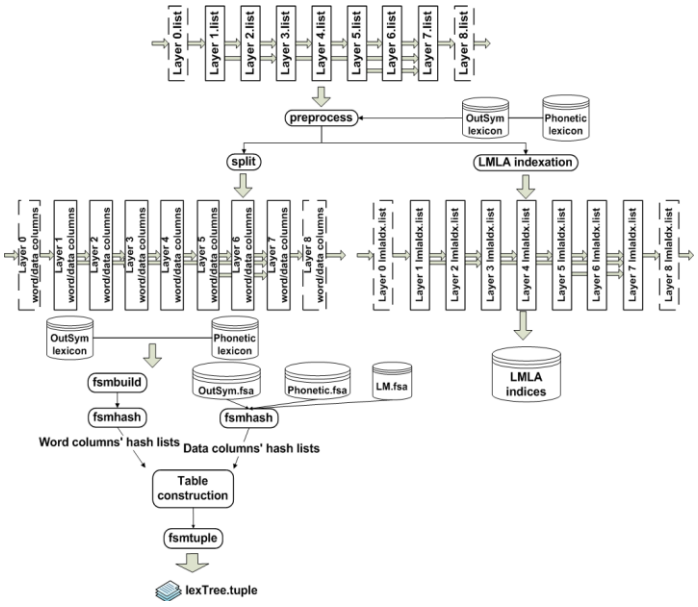

Fig. 10. The construction of the tuple-based LVCSR search network (second step).

In this way there is direct access from each model node to the corresponding HMM model stored within the HMM models' array in the runtime decoder. Context nodes are represented as diphones, and word end nodes with unique node names (layer 7). PPT (layer 6) consists of several sublayers of triphone model nodes, depending on the length of the word. As seen in Figure 9, then follows the creation of links between all these nodes in layers from 0 to 8. In this way, the last nodes within the PPT structure (layer 6) are linked with the corresponding word end nodes in layer 7. The *1*-phoneme nodes in layer 3 are linked with the corresponding word end nodes in layer 7. Furthermore, the context nodes in layer 5 are linked with the starting nodes of the PPT structure, and the model nodes in layer 4 are linked with the context nodes in layer 5. All the word end nodes are further linked with the nodes in layer 8. The final layer 8 is linked back to the corresponding model nodes in layer 0, and the model nodes in layer 0 with the corresponding context nodes in layer 1, and with the silence models in layer 2. Additionally, the context nodes in layer 1 are linked with the silence models. And the silence models in layer 1 are linked with the context nodes in layer 3. It is clear that each of such layers can be represented in the form of "word" and "number" columns. Namely, here word columns represent a node column, and a link column. Additional data (on transitions) can be added in the form of number columns, when needed. Since all layers can be represented in such a way, they can also be compiled into tuple structures. Therefore, all the constructed layers and lexicons are first split into word columns (actually diphones, triphones, ortographic, phonetic transcriptions, output symbols etc.) and translated into corresponding hash-keys (Figure 10). This is performed by using the *fsmbuild* and *fsmhash* tools. Additionally, in layers 0, 3, 4, 5, and 6, phonetic and LM information has to be added in the form of additional data columns (number columns). Next, a table form for each layer is constructed, consisting of several word and number columns (data). Finally, all the tuple structures are constructed and merged by using the *fsmtuple* tool. Additionally, LMLA indexing of all the nodes is performed. In this step all those nodes are numbered, where LMLA calculation has to be performed (the LMLA technique will be presented in the next subsection). The value 0 is assigned only to unique successors in the PPT, since in this case no LMLA calculation is needed. All nodes' LMLA indices are stored as a binary file. Within the runtime system they are loaded, and then directly accessible via hash-keys.

C. Compiling Language Model Look-Ahead Data

Calculating all the possible LM probabilities for all the tokens takes a lot of time and consumes a lot of computational resources. When the lexical network is constructed as a static tuple-based PPT, as described in the previous subsection, word identities can be determined only after there are no more branches in the tree structure. Thus, any inclusion of the language model (LM) probability is delayed until the final nodes are reached. It is well-known that by using LM probabilities in such structure as early as possible, enhances the beam pruning and, therefore, decreases the size of the search-space. This can be achieved by applying so-called language model Look-Ahead techniques. In the literature a number of methods are proposed for managing these calculations [8,16]. The least complex way for reducing the needed number of LM lookups whilst applying LMLA, is to use for the Look-Ahead only unigram probabilities. By using unigrams, the approximation of the best final LM score is less precise, but it becomes possible to integrate the corresponding Look-Ahead scores directly within the PPT. In this case, each node stores a single value: the difference between the best LM score from before and after entering the particular node. In the case of unigrams, these Look-Ahead values can be applied for all tokens, without regard to their *n*-gram history. However, it has been shown that unigram Look-Ahead is outperformed by higher order Look-Ahead systems [8]. A method that can be used for reducing the number of LM lookups has been proposed in [20]. In this case, all those PPT nodes with only one successor node are skipped when calculating the LMLA values. Their decoder used tree copies in order to incorporate the LM probabilities. Furthermore, whenever a new copy is required, the LMLA is performed on demand. In [11] at each PPT node, a special list is stored with all those words that are still reachable from that node. In the cases of small word lists, the Look-Ahead value is calculated exactly (each trigram probability is calculated, and the best one selected). Larger word lists at the PPT root node, are skipped. For all remaining lists, the intersection with the n-gram lists is calculated, before computing the corresponding LMLA values. This approach can save a considerable amount of search-time, especially for those words that do not have a trigram or bigram LM value. The proposed LMLA technique is based on tuple structures. In this approach, Look-Ahead structures are tuples that are constructed off-line. The LVCSR decoder SPREAD does not make tree copies. Instead, LM histories are stored in the tokens and the PPT tuple is shared by all the tokens. In this decoder, the language model knowledge is added to the hypothesis score at the PPT tuple's leaf nodes. Incorporating the LM model at an early stage into the tuple structure, makes it possible to compare and prune the hypotheses based on both linguistic and acoustic evidence.

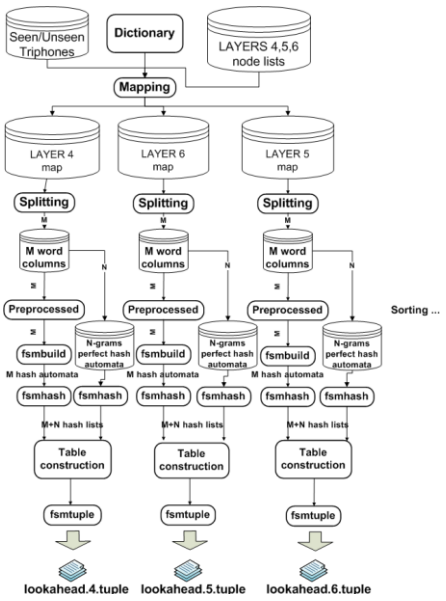

Fig. 11. Compiling LM Look-Ahead tuples for layers 4, 5, and 6.

In the SPREAD LVCSR decoder, the LMLA tuple-based mechanism in the runtime system performs calculations for each token in the tuple, the LM probabilities for all words that are reachable from that token, and temporarily adds the best one to the token's score. When the token reaches a PPT tuple's leaf node, the temporary LM probability is replaced by the probability of the word represented by the PPT tuple's leaf node. Following this procedure, sharper beams can be applied during the pruning so that fewer tokens need to be processed and, consequently, the decoding is speeded up considerably. Each node within the PPT tuple that has more than one successor, or that is a leaf node, is assigned a unique LMLA index. These indices are found in the binary file 'LMLA indices' (Figure 10). The LM Look-Ahead score is computed by finding the maximum of the LM scores over the words in the specific node's list, only when the node's LMLA index value is not 0. The words and the corresponding LM probabilities are accessed via LMLA tuple structures. Nevertheless, in order to minimize the significant amount of redundant computations involved in the LM Look-Ahead, a caching structure is also part of the LMLA process within a runtime system. The caching structure contains the Look-Ahead values for those tokens with a particular language model history. In this way, for each node the maximum LM scores of the possible follow-up words are stored for specific word histories. The LMLA index of a specific node then points to these corresponding LMLA probabilities in cache. Using this method, each node's LMLA probability is exactly calculated once. Therefore, in the case of a cache miss, the probabilities of all the words in the LM for the given word history are computed and stored to the cache. The LM Look-Ahead is applied only in those nodes where the list of possible word identities has changed from that of the previous nodes. Reducing the number of nodes in which LM Look-Ahead is applied also helps to save memory when node level caching is involved. Figure 11 illustrates the compiling of LMLA tuple structures to be used in the decoder. The input represents the seen/unseen triphones' info, the dictionary, and the nodes from layers 4, 5, and 6, where LMLA has to be performed. Seen/unseen triphones' info is needed in order to link the acoustic triphones' nodes in these layers with the dictionary entries, as used in unigrams, and bigrams. After the LVCSR search-network (Figure 10) has been built, the node lists for layers 4, 5, and 6 can be created (consisting of diphones and triphones). The next step is the mapping. Based on the seen/unseen triphones' info, the dictionary, and the node lists, for each layer corresponding maps are constructed, containing all the word identities that are reachable from each node in those layers. These maps are actually tables consisting of diphones or triphones in the first column, and corresponding possible words in the second. The next step is to split each map file into *M* separate word columns (one column contains layers' nodes, and the other corresponding words from dictionary). Next, perfect hash automata are constructed for the *M* separate word columns, using the *fsmbuild* tool. Then all the entries in the *M* word columns are translated into hash-keys by using the *fsmhash* tool. Since the LM Look-Ahead structures are constructed off-line, LMLA values cannot already be stored directly within the LMLA tuple structure. Instead, N data columns are created, containing hash-keys for the corresponding N-grams, by using N-grams perfect hash automata and the *fsmhash* tool. In this way, direct access to LM scores is possible in the online LVCSR decoder, when the LM histories are also known. Now, the corresponding tables for all layers can be constructed, containing *M* separated word columns, and *N* number columns (unigram and bigram hash keys). Finally, the tables are compiled into tuple structures. In this way, three tuple structures are obtained. Within the runtime system they are accessed in layers 4, 5, and 6.

VIII. Results

The LVCSR decoders used today employ acoustic models, pronunciation lexicon, N-gram language models, and other linguistic sources. An approach using efficient and compact tuple structures was proposed in the paper, for a construction of the LVCSR search network. As presented, tuple structures can be implemented as ordinary dictionaries. Namely, the elements within the tuple structures of a given key are concatenated with a selected separator symbol. This also means that a standard implementation of dictionaries can be employed based on perfect hash. The benefits are foremost, the important space savings and higher processing speed (automata), and the compact and reduced size of the tuple structure, especially when the structure of the key can be exploited (depending on the used knowledge sources). In this way, the time needed to load LVCSR search network into the memory is practically instantaneous. Furthermore, fast switching between several applications' specific knowledge sources is possible, since the LVCSR search network is already constructed off-line, and just loaded within the runtime system.

As presented in this paper in detail, application specific ASR knowledge sources can be compiled into tuple-based LVCSR search-network. All the needed steps are accomplished by using several Perl scripts, with proprietary FSM tools, developed in the C++ programming language. The whole procedure is completed within a matter of minutes on a PC with Intel Core 2 Quad CPU, 2.83 GHz, with a 4 GB RAM.

The largest part is spent compiling the *N*-gram language model into the tuple structure. Overall, the whole compiling procedure is simple, fast, without a large memory, or processor requirements. In the experiment, the following ASR knowledge sources were used: context-dependent acoustic models (triphones), a dictionary, and an interpolated bigram language model. The dictionary contained 64K words, and the bigram language model consisted of 64K unigrams, and of approx. 7M bigrams. The proposed methodology for compiling ASR knowledge sources into a tuple structure, can also be used in the same way for higher-order language models (if available), and for other application specific knowledge sources, and languages.

Table 1 presents the statistics about the layers' nodes of the tuple-based LVCSR search network for this speech recognition task. Table 2 then presents statistics about the nodes of the tuple-based language models, and Table 3 presents statistics about the nodes in the tuple-based LMLA structures. These data are based on table forms constructed by using available knowledge sources.

TABLE I. THE LAYERS' NODES IN THE TUPLE-BASED LVCSR SEARCH NETWORK

Layer	0	1	2	3	4	5	6	7	8
Nodes	8,155	650	651	651	7,221	497	1,254,739	64,874	650

TABLE II. THE NODES IN THE TUPLE-BASED LM MODEL

N-gram	1-grams	2-grams
Nodes	64,000	127,696

TABLE III. THE NODES IN THE TUPLE-BASED LMLA STRUCTURES

LMLA	LMLA – layer 4	LMLA – layer 5	LMLA – layer 6
Nodes	71,183	64,457	66,645

TABLE IV. THE COMPACT SIZES OF TUPLES USED FOR LVCSR SEARCH NETWORK

Layer	0	1	2	3	4	5
Tuple	304kB	3.93kB	11.2kB	274kB	182kB	81.8kB
Layer	6	7	8			
Tuple	5.87MB	380kB	47.1kB			

TABLE V. THE COMPACT SIZES OF TUPLES USED FOR LM MODEL

N-gram	1-grams	2-grams
Tuple	812kB	63.4MB

TABLE VI. THE COMPACT SIZES FOR TUPLES FOR LMLA DATA

LMLA	LMLA – layer 4	LMLA – layer 5	LMLA – layer 6
Tuple	9.43MB	438kB	440kB

All the table forms additionally contain several data columns (number columns) that are used within the ASR system. The tables 4-6 then represent the achieved compact sizes of the tuples after compiling constructed table forms. The sizes reported in the tables are the sizes of the final compiled files. The overall size of the merged tuple structure loaded for the specific task by the SPREAD LVCSR decoder is 81.234 MB for the 64k LVCSR task.

The same task was also tested by HDecode [27]. In the case of HDecode, the loading of knowledge sources prepared in their format and construction of internal ASR structures, took 50 times longer (since all the structures for the LM, LMLA and LVCSR search-network has to be constructed during initialisation). Furthermore, a set of 100 audio files was recognized by using both decoders in order to evaluate whether the tuple-based decoder also showed any benefits regarding the processing speed. In both systems the same configuration was performed in order to compare the obtained results. In the case of the SPREAD LVCSR decoder, approx. 20% higher processing speed was achieved, without loss of recognition accuracy. All the experiments were performed on a PC with Intel Core 2 Quad CPU, 2.83 GHz, with a 4 GB RAM.

IX. CONCLUSION

This paper presented the novel design of a LVCSR decoder engine, named SPREAD. This LVCSR decoder is based on a time-synchronous beam search approach. The ASR search network includes statically expanded cross-word triphone contexts. An approach using efficient tuple structures was proposed and presented, for constructing a complete ASR search-network. These data structures were motivated by practical applications in speech and language processing. The used technique for compact representation of tuple structures can be seen as an application and extension of perfect hashing by means of finite-state automata. Therefore, the benefits are foremost the important space savings and higher processing speed. Furthermore, the advantage of the proposed LVCSR decoder implementation based on tuple structures is the compact and reduced size of the tuple structure, especially when exploiting the structure of the key (*n*-tuples of strings). Therefore, the time needed to load an ASR search-network into the memory is also significantly reduced. Further, in the paper the complete methodology of compiling general ASR knowledge sources into a tuple structure (representing an ASR search-network) was proposed and presented. It has been shown that ASR knowledge sources can be implemented by ordinary dictionaries, where the elements in the tuple of a given key are concatenated with a specific separator symbol of our choice. Therefore, a standard implementation of dictionaries can be employed, typically a hash table or perfect hash.

Furthermore, the beam search was enhanced with a novel implementation of bigram language model Look-Ahead technique, by using a tuple structure and a caching scheme. The SPREAD LVCSR decoder is based on a token-passing algorithm and is able to restrict search-space by several types of token pruning. By using the presented language model look-ahead (LMLA) technique, it is possible to increase the number of tokens that can be pruned without any decoding precision loss.

REFERENCES

[1] Aubert, X. An overview of decoding techniques for large vocabulary continuous speech recognition. Computer Speech & Language, Volume 16, issue 1, pp. 89-114, 2002.

[2] Creutz, M., Hirsimäki, T., Kurimo, M., Puurula, A., Pylkkönen, J., Siivola, V., Varjokallio, M., Arisoy, E., Saraçlar, M., Stolcke, A. Morph-based speech recognition and modeling of out-of-vocabulary words across languages. ACM Trans. Speech Lang. Process. Volume 5, issue 1, Article 3 (December 2007), pp. 1-29, 2007.

[3] Daciuk, J., van Noord, G. Finite automata for compact representation of tuple dictionaries, Theoretical Computer Science, Volume 313, Issue 1, pp. 45-56, 16 February 2004.

[4] Dixon, P., Caseiro, D., Oonishi, T., Furui, S. The Titech Large Vocabulary WFST Speech Recognition System. In Proc. ASRU, pp. 1301-1304, 2007.

[5] Evermann, G., Woodland, P. C. Design of fast LVCSR systems. In Proc. ASRU'03, 2003, pp. 7–12, 2003.

[6] Fujii, Y., Yamamoto, K., Nakagawa, S. Large vocabulary speech recognition system: SPOJUS++. In Proceedings of the 11th WSEAS international conference on robotics, control and manufacturing technology, and 11th WSEAS international conference on Multimedia systems & signal processing (ROCOM'11/MUSP'11). S. Chen, Nikos Mastorakis, Francklin Rivas-Echeverria, and Valeri Mladenov (Eds.). World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, pp.110-118, 2011.

[7] Hoffmeister, B., Heigold, G., Rybach, D., Schlüter, R., Ney, H. WFST Enabled Solutions to ASR Problems: Beyond HMM Decoding. IEEE Transactions on Audio, Speech, and Language Processing. Volume 20, number 2, pp. 551-564, February 2012.

[8] Huijbregts, M., Ordelman, R., de Jong, F. Fast N-gram Language Model Look-Ahead for Decoders with Static Pronunciation Prefix Trees. In Interspeech 2008, 9th Annual Conference of the International Speech Communication Association, pp. 1582-1585, Brisbane, Australia, September 22-26, 2008.

[9] Lee, A., Kawahara, T., Shikano, K. Julius - an open source real-time large vocabulary recognition engine. In proceedings of Eurospeech 2001, pp. 1691-1694, 2001.

[10] Liu, X., Gales, M. J. F., Woodland, P. C. Improving LVCSR system combination using neural network language model cross adaptation. In Interspeech 2011, pp. 2857-2860, Florence, Italy, August 2011.

[11] Massonie, D., Nocera, P., Linares, G. Scalable language model look-ahead for LVCSR. In proceedings Interspeech 2005, Lisbon, Portugal, pp. 569-572, 2005.

[12] Mohri, M., Pereira, F., Riley, M. Weighted finite-state transducers in speech recognition. In: Computer Speech and Language 16, pp. 69-88, 2002.

[13] Moore, D., Dines, J., Magimai Doss, M., Vepa, O., Cheng, O., Hain, T. Juicer: AWeighted Finite State Transducer Speech Decoder. In Proc. Interspeech, pp. 241-244, 2005.

[14] Moore, D., Dines, J., Magimai-Doss, M., Vepa, J., Cheng, O., Hain, T. Juicer: A Weighted Finite-State Transducer Speech Decoder. In MLMI'06, 3rd joint Workshop on Multimodal Interaction and Related Machine Learning Algorithms, pp. 285-296, 2006.

[15] Nguyen, P. Techware: Speech recognition software and resources on the web. IEEE signal processing magazine. Volume 26, number 3, pp. 102-105, 2009.

[16] Nolden, D., Schlüter, R., Ney, H. Acoustic Look-Ahead for More Efficient Decoding in LVCSR. In Interspeech 2011, pp. 893-896, Florence, Italy, August 2011.

[17] Novak, J. R., Dixon, P. R., Furui, S. An empirical comparison of the t3, juicer, HDecode and sphinx3 decoders. In Interspeech 2010, pp. 1890-1893, 2010.

[18] Novak, J. R., Minematsu, N., Hirose, K. Painless WFST cascade construction for LVCSR – transducersaurus. In Interspeech 2011, pp. 1537-1540, Florence, Italy, August 2011.

[19] Novak, J. R., Minematsu, N., Hirose, K. Open Source WFST Tools for LVCSR Cascade Development. Finite-State Methods and Natural Language Processing, 9th International Workshop, FSMNLP 2011, pp. 65-73, Bois, France, July 12-16, 2011.

[20] Ortmanns, S., Ney, H., Eiden, A., Coenen, N. Look-ahead techniques for improved beam search. In proceedings of the CRIM-FORWISS Workshop, Montreal, pp. 10-22, 1996.

[21] Parada, C., Dredze, M., Sethy, A., Rastrow, A. Learning Sub-Word Units for Open Vocabulary Speech Recognition. Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, 2011.

[22] Pylkkönen, J. An Efficient One-pass Decoder for Finnish Large Vocabulary Continuous Speech Recognition, Proceedings of the 2nd Baltic Conference on Human Language Technologies (HLT'2005), Tallinn, Estonia, pp. 167-172, April 4-5, 2005.

[23] Rojc, M., Kačič, Z. Time and Space-Efficient Architecture for a Corpus-based Text-to-Speech Synthesis System, Speech Communication, Vol. 49 (3), pp. 230-249, 2007.

[24] Rybach, D., Hahn, S., Lehnen, P., Nolden, D., Sundermeyer, M., Tüske, Z., Wiesler, S., Schlüter, R., Ney, H. RASR - The RWTH Aachen University Open Source Speech Recognition Toolkit. In IEEE Automatic Speech Recognition and Understanding Workshop (ASRU), Hawaii, USA, December 2011.

[25] Sixtus, A., Ney, H. From within-word model search to across-word model search in large vocabulary continuous speech recognition. In: Computer Speechamd Language 16, pp. 245-271, 2002.

[26] Walker, W., Lamere, P., Kwok, P., Raj, B., Singh, R., Gouvea, E., Wolf, P., Woelfel, K. Sphinx-4: A flexible open source framework for speech recognition. Sun Microsystems Technical Report, No. TR-2004-139, Sun Microsystems Laboratories, 2004.

[27] Young, S., Everman, G., Kershaw, D., Moore, G., Odell, J., Ollason, D., Valtchev, V.,Woodland, P. The HTK Book. Cambridge University Engineering, 2006.