Optimization of Weighted Finite State Transducer for Speech Recognition


Published in:
IEEE Transactions on Computers

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Publisher rights
© 20xx IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.
Optimization of Weighted Finite State Transducer for Speech Recognition

Louis-Marie Aubert, Roger Woods, Senior Member, IEEE, Scott Fischaber Member, IEEE, and Richard Veitch

Abstract—There is considerable interest in creating embedded, speech recognition hardware using the weighted finite state transducer (WFST) but there are performance and memory usage challenges. Two system optimization techniques are presented to address this; one approach improves token propagation by removing the WFST epsilon input arcs; another one-pass, adaptive pruning algorithm gives a dramatic reduction in active nodes to be computed. Results for memory and bandwidth are given for a 5000 word vocabulary giving a better practical performance than conventional WFST; this is then exploited in an adaptive pruning algorithm which reduces the active nodes from 30000 down to 4000 with only a 2% sacrifice in speech recognition accuracy; these optimisations lead to a more simplified design with deterministic performance.

Index Terms—Embedded processors, memory organization, speech recognition, WFST.

I. INTRODUCTION

W ith the evolution in information and communications technology (ICT), easier forms of interaction such as Automatic Speech Recognition (ASR) have become important. Many solutions such as Nuance’s Dragon Natural Speaking technology are server-based, but increasingly there has been interest in embedded system solutions such as Field Programmable Gate Array (FPGA) [1], [2], [3], [4] and Application Specific Integrated Circuit (ASIC) [5] solutions; a faster than real-time performance using multi-FPGAs has been reported [1]. The various information sources for these speech recognition systems, are built up dynamically and on-demand, however, this is problematic for embedded systems as it means the performance will vary depending on the speech that is to be decoded.

An alternative approach is to use the weighted finite state transducer (WFST) method [6]; this involves creating a single, larger data structure that results in a system realization that does not need modification. The key advantage is that the underlying hardware remains static as changes in the language model just involves the loading of a new data file. However, the flattening of the multiple data sources into a single transducer results in an increase in the epsilon arcs; this reduces the overall number of arcs but increases the number of nodes. When decoding the network, more transitions are invoked and thus memory accesses are increased which is not good for embedded system implementation. Some solutions have been proposed to address these challenges; in [5], they tolerate the issue and build a system with a variable memory access time whereas in [3], SRAM and DRAM memory solutions are proposed.

The proposed approach here removes epsilon arcs by recasting the WFST (reducing the bandwidth required) and using a new means of pruning to reduce token memory storage giving a small loss in speech recognition. In addition to relaxing constraints on memory size and bandwidth, this has also positive implications for embedded system in terms of real-time operation and energy consumption. The contributions of the work are as follows:

- A technique to remove epsilon arcs resulting in a more predictable memory access pattern.
- Pruning approaches to reduce the number of active nodes required from 30000 to 4000 with a very small sacrifice in speech recognition accuracy.

The paper is organized as follows. Section II introduces ASR and in section III, WFST-based ASR is presented along with the major challenges for its embedded system implementation. Section IV outlines the approaches used to overcome these problems; conclusions are then given in Section V.

II. AUTOMATIC SPEECH RECOGNITION

Speech recognition involves: feature extraction which creates Acoustic Observation Vectors (AOVs) based on the speech input; acoustic scoring which evaluates the feature vectors against acoustic models (represented as Gaussian Mixture Models or GMMs) in order to identify the most probable pronounced phonemes (represented as Hidden Markov Models or HMMs); and a back-end search which builds words and sentences from the phoneme candidates based on a set of rules compiled from lexicons and language models. As HMMs cannot be identified with enough reliability, the backend search is needed but is restricted by the two additional knowledge sources, the lexicon and the language model. This is traditionally implemented by the technique of token propagation on HMMs which is dynamically linked to the other functions according to the input speech and the knowledge sources.

A. HMM-based Implementations

Profiling of the Sphinx 3.0 HMM-based speech recognition shows that GMM scoring dominates the complexity of
small vocabulary speech recognition; however, the back-end search becomes predominant for large lexicons and complex language models [1]. A hardware version of the back-end search of Sphinx [7] showed that the memory requirements exceed current embedded systems and that on-chip SRAM accounted for 36% of the total power consumed by the back-end search. Whilst dynamic composition presents several difficulties for embedded systems, the WFST structure offers considerable potential for achieving efficient regularized decoding architectures by enabling the creation of the network in an off-line manner [6], [8].

III. THE CHOICE OF THE WFST FRAMEWORK

A. WFST background

The WFST framework [6] is a Mealy finite state machine representation where knowledge sources are incorporated into a single unified representation, allowing all path redundancies to be removed by the steps of determinization and minimization [8]. It is a network consisting of nodes linked by arcs where each arc consists of an input, an output, and a transition probability. All three sources, the language or ‘G’ model, the lexicon, ‘L’, and HMM or ‘H’ state sequences, can be modeled as a WFST:

- Language models are represented as Weighted FSA (Finite State Accumulator) i.e. an WFST with inputs identical to outputs on every arc. Fig. 1(a) gives a very simple language model for a four-word vocabulary. For larger vocabulary, n-gram language models are stochastic models learnt from data, giving the probability of occurrence of every word and sequence of words.
- The lexicon WFST is built from a dictionary that contains all words with their phoneme transcription. The epsilon arc (labeled ‘-:-’) is the closure that is applied to the transducer L, so that every word in the dictionary can follow any other word.
- The ‘H’ WFSTs has GMM’s references on arcs’ inputs and is illustrated on Fig. 1(c). When an arc is taken, the input that corresponds to a GMM is evaluated for the current AOV with the resulting Gaussian probability multiplied by the fixed transition probability of the arc. The final arc of the HMM represents the output phoneme that the HMM characterizes.

The final WFST is achieved by combining these WFSTs as illustrated for the LoG as shown in Fig. 2(a). Using determination and minimization to remove any redundancy, a much more compact equivalent WFST results (see Fig. 2(b)). The same procedure is applied between H and LoG to form a compact WFST graph HoCoLoG (Fig. 2(c)). In practice though, HMMs rather than phonemes, are made to characterize triphones for robustness in continuous speech, natural language. A context dependency WFST network ‘C’ converting triphones into sequences of phonemes is used to build the HoCoLoG, where arcs take GMMs as output words.

B. WFST Decoding - Search and pruning techniques

In the WFST context, the search of the best word sequence \( W^* \), can be written simply as:

\[
W^* = \arg \max_{S \rightarrow W} \prod_{t=1}^{T} g_{s_t}(x_t) b_{s_t,s_{t+1}}
\]  

where \( S \rightarrow W = s_1 \ldots s_t \ldots s_T \) represents the WFST state sequence outputting the word sequence, \( W \), and the terms, \( b_{t,j} \), are the transition probability of the WFST arcs. WFST decoding uses token propagation on a fully composed and optimized precomputed network. By taking the minus-log, the probabilities of equation (1) can be simplified as shown:

\[
W^* = \arg \min_{S \rightarrow W} \sum_{t=1}^{T} \left[ c_{g}(s_t(x_t)) + c_{b}(s_{t},s_{t+1}) \right] 
\]  

where the Gaussian cost, \( c_{g}(x) \), is defined as \(- \log(g(x))\) and the arc costs, \( c_{b} \), as \(- \log(b)\).

Tokens are propagated from node to node along the arcs of the WFST at every frame. Each token contains the node on which it is located (the node is then called an active node), a reference to the word sequence output so far, and the running cost (the cumulative sum of the terms in bracket in equation 2)). The minimum in equation (2) is not known until the very end of the utterance at time \( t = T \), so the system must use multiple tokens to explore and track multiple hypotheses. Fig. 3 illustrates the propagation of tokens for one frame; the word sequence history is kept in a linked list called Word Link Record (WLR).

C. Implementation thoughts

The WFST framework offers a uniform way of representing the different knowledge sources; it allows the
decoding of speech with a simplified decoder. Whilst the token propagation process is straightforward, it is complicated by the presence of so-called epsilon input arcs. They are introduced at different stages during the WFST composition, predominantly by the back-off states in the language model [8] and the silence/short pause acoustic models; they represent a non-physical link between two nodes and belong to the same audio frame. For a token to be propagated during one frame, it must go therefore through all epsilon arcs until it is eventually propagated on a non-epsilon arc. This means that the time for decoding will depend on the WFST structure and may result in a variance in access time; this can be significant depending on how the data is stored. In [3] and [5], this issue is accepted and taken into consideration in the system design.

The second key aspect is to limit the size of the token memory. This is usually done by using a pruning technique that traditionally requires a trade-off between computational power and recognition accuracy. By exploiting the organization of the data in the WFST as a result on the initial work, the token memory size is able to be greatly reduced without compromising speech recognition accuracy.

IV. ALGORITHM OPTIMIZATION AND IMPLEMENTATION

To quantify the performance of our approach, the Nov’92 WSJ task and acoustic models from the work in [9] were used. It was trained with 14 hours of speech and HMM states were clustered into 3825 distinct acoustic models; the WFST is built from word-internal triphones with a 4987 word lexicon and a bi-gram language model. This gives a fully composed WFST HoCoLoG with 1,577,629 nodes and 5,075,857 arcs. Naturally, arcs have 3825 distinct inputs corresponding to acoustic models and 4987 distinct outputs corresponding to words.

A. Epsilon input arcs in the WFST graph

Epsilon arcs present two major difficulties for the token propagation realization. Firstly, they preclude advance loading of nodes and arcs from the WFST memory as they are discovered only as tokens are propagated along the epsilon arcs. As a result, the WFST memory is accessed a number of times in order to load small quantity of data and does not easily permit task pipelining. Secondly, the computation time is unpredictable as the total number of arcs and the resulting number of tokens are not known in advance; this complicates further the pruning of the tokens.
Fig. 4: Recursive function to propagate a token in a WFST with $\epsilon$ arcs.

Whereas caching intermediate results can limit the negative effect of epsilon arcs on the performance of the system [5], we show here that the complete removal of these arcs can simplify and speed up the token propagation.

To propagate tokens in a WFST that contains epsilon arcs, all branches from the active nodes must be explored until non-epsilon arcs are reached. This is typically implemented via a recursive call to a function that propagates on one arc such as that shown in Fig. 4. A temporary WLR needs to be dynamically created to account for cases where multiple outputs are encountered during the propagation of tokens on a series of epsilon arcs.

Even though the WFST is a precomputed network of nodes and arcs, the computations for epsilon removal need to be carried out on-the-fly which is challenging for real-time operation. This can be avoided by applying an off-line technique (Fig. 5) to create an epsilon-free WFST graph. A one-step propagation is applied to each node of the original WFST using the recursive function `Propagate_on_arc` (Fig. 4) meaning that all outgoing arcs from each node are then epsilon free. Part of the process is the removal of arcs and nodes that can no longer be accessed by any token during the decoding e.g. node 3 in Fig. 5. This means that the resulting graph can then be decoded by the simple implementation of token propagation as in Fig. 3.

Since every arc has a non-epsilon input, the depth of the propagation at every frame is always one which means that the arcs and nodes to be loaded from the WFST memory are directly known from the list of active nodes. Whereas the process of removing epsilon inputs can create situations where multiple words are output on a single arc, these situations can be prevented by pushing outputs on nearby arcs during the creation of the epsilon-free WFST. It is easy to comprehend that it is always possible to take more than one input to generate a word, no matter how the WFST graph is set up.

The epsilon removal increases the number of arcs, doubling to 11,069,000 for our example with the number of nodes reduced to 1,546,000. However, when nodes and arcs are loaded as requested by the token propagation, bandwidth requirements are reduced. Epsilon arcs introduce an overhead of 30% and 5% respectively, in the loading of the number of nodes and arcs. As an illustration, a token on node 0 that is propagated to nodes 1, 2, and 4 as in Fig. 5, requires the loading of 3 nodes and 5 arcs in the original WFST, whereas only 1 node and 3 arcs need to be loaded in the epsilon-free version of the same WFST. In our configuration, up to 50 nodes needs to be loaded to propagate one token whereas a epsilon-free WFST allows each token to be propagated by consistently loading only one token.

In practice, the bandwidth overhead penalizing the WFST approach with epsilon arcs can be reduced by caching. Because tokens reflect the multiple hypotheses for the decoded speech, they are located locally in the WFST, so the nodes and arcs that need to be loaded is substantially smaller; this is exploited by a preloading stage of WFST based on active nodes as described by the data flow graph in Fig. 6.

In the case of a WFST with epsilon inputs, the dashed box ‘load WFST data’ in Fig. 6 requires the actual propagation of tokens to explore epsilon arcs which requires a second pass. In the case of epsilon-free WFST, the dashed box ‘load WFST data’ on Fig. 6 is replaced by a straightforward sequence of Load node followed by Load arcs without the conditional loop; nodes and arcs are simply identified from a list. The caching technique gives an advantage to the original WFST in term of raw bandwidth as shown in Table I, where nodes and arcs are stored as 64 and 96 bits respectively. Note that the bandwidth can be further reduced if nodes and arcs can be cached during two consecutive frames (Table I(b)).

The epsilon removal approach reduces the number of nodes to be loaded by 16% in our tests. This means that the number of accesses is reduced by the same amount as the arcs belonging to a node are stored consecutively.
TABLE I: Requirements of WFST loading with caching of nodes and arcs:

(a) Nodes and arcs are cached during one frame.

<table>
<thead>
<tr>
<th></th>
<th>Original WFST</th>
<th>Epsilon free WFST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of nodes</td>
<td>1025</td>
<td>860</td>
</tr>
<tr>
<td>Number of arcs</td>
<td>4540</td>
<td>5670</td>
</tr>
<tr>
<td>Total bandwidth</td>
<td>50</td>
<td>60</td>
</tr>
</tbody>
</table>

(b) Nodes and arcs are kept loaded during two consecutive frames.

<table>
<thead>
<tr>
<th></th>
<th>Original WFST</th>
<th>Epsilon free WFST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of new nodes</td>
<td>155</td>
<td>140</td>
</tr>
<tr>
<td>Number of new arcs</td>
<td>585</td>
<td>810</td>
</tr>
<tr>
<td>Total bandwidth</td>
<td>6.5</td>
<td>8.5</td>
</tr>
</tbody>
</table>

Fig. 6: Data flow graph of WFST preloading

in memory, and the start address and the number of arcs associated are known only when nodes are read. So the epsilon removal allows fewer accesses to memory and data is loaded from larger blocks in the memory.

The basic analysis is that the epsilon removal technique results in a dramatic increase in the number of arcs but this is offset by a number of factors related to the practical realization of the system:

- On-chip memory size and bandwidth can be lowered as the number of nodes has actually been reduced as shown in Table I(a); whilst the arcs have increased, the epsilon-free WFST will never require them to be reloaded which can result in a considerable delay in DRAM technologies particularly if the reload involves loading a new row.
- Off-chip memory bandwidth can be reduced as the figures in Table I(a) represent average cases; this can vary in the original WFST but not in the epsilon-free case.

Finally, the increased WFST graph size can be offset by factorization techniques where the most frequent sequences of nodes and arcs are extracted from the main WFST to form a subset of WFST paths [8]. These paths are then re-composed with the main WFST graph at run-time at low cost; this has the added benefit of simplifying the look-ahead techniques for preloading subsets of arcs and nodes from the off-chip memory.

B. Pruning

The second major difficulty of token propagation is the implementation of pruning. Because the number of hypotheses grows exponentially as we go through the WFST network, the best tokens must be selected and the least probable hypotheses then discarded. An approach for pruning is presented here which benefits from the reorganization of the WFST graph in the previous section.

Two techniques are commonly used, the beam and the histogram pruning (Fig. 7). Beam pruning discards tokens with higher costs than the best one for a fixed beam width as shown in Fig. 8. In order to avoid the storage of all tokens, the pruning threshold can be based on the current best, thus allowing the beam pruning to be done live. However, if best tokens are not propagated first, the beam pruning threshold is higher than desired and unpromising tokens are not pruned. Moreover, as beam pruning fixes a boundary on the cost of tokens, the number of output tokens can be still large. The issues are that the memory size is determined by worst case scenarios and the tokens with higher cost are unnecessarily kept alive and propagated further.

To address this latter issue, histogram pruning which generates a histogram of output tokens costs, is commonly applied; this is used to set the maximum cost of the input tokens at the next frame so that only a predefined number of tokens have a cost lower than this threshold. This requires a second pass through all tokens and results in the unnecessary storage of a large number of tokens; one approach is to tighten the beam threshold but this reduces the resulting recognition accuracy while not making the most efficient use of available memory.

Instead of reducing the beam width for all frames, an adaptive beam pruning threshold can be used which is decreased when tokens are too numerous, and increased, when too few. This has the advantage of requiring only the storage of the wanted tokens (Fig. 8(c)). For example on our test set, if we set the maximum number of tokens at 4000 (i.e. 4000 tokens are propagated during the next frame), a maximum of 30000 tokens have to be evaluated during a frame. Even when evaluating so many tokens, an adaptive pruning algorithm allows the memory to be dimensioned to store only 4000 tokens.
for all $tk \in \text{output tokens}$ do
\{Beam pruning\}
    if $tk \rightarrow \text{cost} < \text{best\_cost}$ then
        $\text{best\_cost} \leftarrow tk \rightarrow \text{cost}$
    else
        if $tk \rightarrow \text{cost} > \rho_{\text{beam}}$ then
            Prune $tk$
        else
            Keep $tk$
            $\text{bin} \leftarrow \text{determine\_bin}(tk \rightarrow \text{cost})$
            $\text{hist}(\text{bin}) \leftarrow \text{hist}(\text{bin}) + 1$
        end if
    end if
end for

\{Histogram pruning\}
Determine $\rho_{\text{hist}} = \text{threshold}(\text{bin})$ so that
\[ \text{bin} = \max \arg_b \left\{ \sum_{i=1}^{b} \text{hist}(i) < N_{\text{max}} \right\} \]
for all $tk \in \text{input tokens}$ do
    if $tk \rightarrow \text{cost} > \rho_{\text{hist}}$ then
        Prune $tk$
    else
        Propagate $tk$
    end if
end for

Fig. 7: Beam and Histogram pruning algorithm.

Next, we first devise an algorithm that efficiently implements adaptive beam pruning and then we present an alternative token sorting approach that extends the memory savings by preventing the propagation of unpromising tokens.

1) Adaptive pruning: For most efficient use resources, the number of tokens retained should equal the number that the system can store at every frame; however, the number of output tokens will vary with the ambiguity of the input speech. Without a pruning beam threshold, the number of output tokens, $n_{\infty, t+1}$ (frame $t+1$), depends on the number of input tokens, $n_{b, t}$ (frame $t$), and the number of arcs, $a_i$ and is given as:
\[ n_{\infty, t+1} = \sum_{i=1}^{n_{b, t}} a_i \]
(3)
This effectively involves every node where input tokens of index, $i$, are located. The number of output tokens, $n_{b, t+1}$, whose cost is under the threshold, $b$, naturally depends on the initial costs, $c_i$, and propagation cost, $p_{i,j}$, i.e. the sum of the static arc cost and the Gaussian cost. It is given by:
\[ n_{b, t+1} = \sum_{i=1}^{n_{b, t}} \sum_{j=1}^{a_i} \left[ c_i + p_{i,j} < b \right] \]
(4)
where we adopt the Iverson bracket notation:
\[ [A] = \begin{cases} 1 & \text{if } A \text{ is true;} \\ 0 & \text{otherwise.} \end{cases} \]
For every frame, we want to set the threshold, $b$, so that the number of tokens retained is under the threshold but also below $N$, the number of tokens the system can hold. Thus, this is given as:
\[ n_{b, t} \leq N \ \forall \text{ frame } t \]
(5)
From equation (4), it is clear that tokens need to be propagated to the next frame to allow determination of the exact number of tokens to be kept for a given threshold, $b$, as costs, $c_{i,j}$, need to be calculated and sorted. This is achieved by histogram pruning, but if we want to avoid any degree of sorting and unnecessary token storage, the threshold needs to be set before tokens are propagated so that tokens are pruned live in one pass as the resulting number of tokens approaches $N$. The difficulty is that before the $n_{b, t}$ tokens are propagated to the next frame, a definite value of $n_{b, t+1}$ cannot be evaluated but only estimated; this we do statistically.

The analysis of random variables, $c_j$ and $p_{i,j}$, shows that these are dependent, so $c_{i,j} = c_i + p_{i,j}$. Assuming for now that we know $\{ a_i \}$, we can then formulate an estimate of $n_{b, t+1}$ by integrating over all possible values of $c_{i,j}$, weighted by their probability, so that:
\[ n_{b, t+1} = \sum_{i=1}^{n_{b, t}} \sum_{j=1}^{a_i} \left[ c_{i,j} < b \right] \int_0^\infty c_{i,j} p_c(c_{i,j}) dc_{i,j} \]
(6)
where $p_c(c)$ is the probability density function of the random variable $c$. It is fair to assume that the distribution of $c_{i,j}$ does not actually depend on $i$ and $j$, the indexes of the tokens and outgoing arcs respectively, so equation (6)
simplifies to:
\[ \tilde{n}_{b,t+1} = \sum_{i=1}^{n_{b,t}} a_i \int_{0}^{b} p_c(c) dc \]  
(7)

The integral \( \int_{0}^{b} p_c(c) dc \) is the cumulative distribution function (CDF) of \( c \), noted \( P_c(b) \), thus a simple estimator of \( n_{b,t+1} \) is:
\[ \tilde{n}_{b,t+1} = \sum_{i=1}^{n_{b,t}} a_i P_c(b) \]  
(8)

The following assumptions are made.
- The number of input token \( n_{b,t} \) is easily known.
- The sum of arcs from all nodes where tokens are currently located, i.e. \( \sum_{i=1}^{n_{b,t}} a_i \), is easily computed when nodes are preloaded from the WFST graph, particularly as the epsilon arcs have been removed beforehand; alternatively, \( \sum_{i=1}^{n_{b,t}} a_i \) can be approximated by the product \( n_{b,t} a \) where \( a \) is an estimated average number of outgoing arcs per node.
- The CDF \( P_c(b) \) can be evaluated off-line, allowing the generation of equation (8) which can be precomputed and resulting values in a fixed look-up table (LUT). Due to great variability from one setup to another, the use of a predetermined function offers poor performance. Instead, we evaluate the CDF live during the token propagation.

The CDF is defined as \( P_c(b) = P(c < b) \) and can be estimated from previous tokens propagation by \( \tilde{P}_c(b) = \frac{n_{b,t}}{n_{tot}} \), where \( n_b \) is the number of tokens whose cost is lower than \( b \) and \( n_{tot} \) is the total number of tokens propagated. A recursive version allows the estimation to be made live during the propagation:
\[ \tilde{P}_{k+1}(b) = \begin{cases} \frac{k \tilde{P}_c(b) + 1}{k+1} & \text{if the new token cost } c_{k+1} < b; \\ \tilde{P}_c(b) & \text{otherwise.} \end{cases} \]  
(9)

It is made dynamic by introducing a gradual loss of old values by simply fixing \( k + 1 \) to a fixed value \( K \):
\[ \tilde{P}_{k+1}(b) = \begin{cases} \frac{\tilde{P}_c(b) - K \tilde{P}_c(b)}{K} & \text{if } c_{k+1} < b; \\ \frac{\tilde{P}_c(b) - K \tilde{P}_c(b)}{K} & \text{otherwise.} \end{cases} \]  
(10)

In fixed point integer, the formulae are implemented by choosing \( K \) to be a power of 2 resulting in cost-effective hardware; they are estimated at several costs between 0 and the required maximum beam pruning width. Results are maintained in a CDF look-up-table (CDF-LUT) and used to continuously adapt the threshold by incrementing or decrementing its value. If the estimated number of output tokens is higher than the maximum, \( N_{\text{max}} \), the beam pruning threshold is changed to the next lower value in the CDF-LUT whereas if lower, the threshold is increased. Because the overhead of the adaptive algorithm is low, these adjustments can be often made. Fig. 9 gives the pseudocode for this adaptive pruning algorithm where the estimate and the threshold are updated every new output token. However, as shown in Fig. 10, a lower update rate can be adopted without significantly affecting the recognition accuracy.

A simpler adaptive pruning algorithm [2] works by incrementing or decrementing the pruning threshold by a fixed step size at every frame based solely on the number of input tokens. This algorithm has the advantage of a low overhead, but the low update rate plus the failure to take token costs into consideration has consequences in that the adaptive algorithm degrades recognition accuracy performance. For our test set, this gives a maximum recognition accuracy of 78.5% compared to 88.3% with our algorithm.

Compared to histogram pruning, adaptive pruning saves the storage of unnecessary tokens; however, a large number of tokens are still propagated which are eventually discarded. The solution is to sort the tokens on-the-fly in increasing cost order; the selection of the best tokens or pruning, is then trivial. In [10], the histogram threshold is optimized through an analysis of the training data set in order to derive a set of pruning thresholds. Whereas it is particularly well suited to speaker dependent tasks, it offers little benefit in the more challenging speaker independent case. We have explored techniques where skip lists and parallel sorting by sort-trees are the most promising for exhaustive sorting in hardware. However, studies showed that a non-exhaustive or coarse sorting offers very similar recognition performances for an even more efficient, em-
bedded system realization.

2) Pruning by token sorting: We have devised an efficient data structure that allows tokens to be sorted 'live' based on their cost between bins of a histogram. This does not require a second pass and only the desired tokens are stored: the limit is applied as part of the propagation rather than applied to the input tokens at the start of the next frame. The list of tokens are stored in an array of structure tokens. In addition to the active node where the token is located, the running cost and the entry in the WLR, the structure token comprises a pointer to the next token belonging to the same sorting sub-division. A list of tokens is thus created comprising of a set of $F$ linked-lists, where $F$ is the number of sorting sub-divisions.

This data structure gives the flexibility to add or remove tokens in any sorting sub-division. During the propagation, tokens that are not discarded by beam pruning are either inserted or dropped (see Fig. 11). Fig. 12 depicts the list of tokens sorted into sub-divisions. When the maximum number of stored tokens is reached (Fig. 12(b)), tokens belonging to the sub-division with the highest cost (indexed $f_{top}$) are replaced by tokens whose cost are lower.

This mechanism allows the system to store a reduced number of tokens at any stage of the propagation as in the adaptive pruning algorithm. The main benefit of sorting tokens is being able to propagate them in order of increasing cost during the next frame, thus ensuring that the first tokens after the propagation in the frame are also the best tokens for the subsequent frame; this results in few replacements without a major impact in recognition accuracy (see Table II). Token propagation in order of increasing cost allows the propagation during a frame to be stopped early, without affecting recognition accuracy.

As linked-list pointers are used, 4000 tokens with a 12-bit pointer requires an additional 48 kbit as compared to the storage of only 600 of 80-bit tokens but this is not enough to maintain the same accuracy (Table II). In a system with 50000 where 60000 tokens could be required, not using our data structure would only allow the additional storage of 12000 tokens. Moreover, the savings offered by sorting tokens extends beyond the token storage only. The list of active nodes and the list of N-best pointers have a size directly linked to the number of propagated tokens. In turn, the reduction in the number of active nodes means fewer nodes and arcs need to be loaded from the WFST memory, thus reducing the memory bandwidth and cache size. Whilst this reduces tokens and thus memory, it does not greatly decrease the number of Gaussian calculations required. In our analysis, the reduction of the maximum of active nodes from 30000 to 400 reduces the number of distinct Gaussian calculations required by only 5% on average; this is due to the fact that the 4000 tokens encapsulate all of the best hypotheses and the additional tokens reuse the same Gaussian results.

C. Comparative research

Dixon [11] describe a similar WFST system with general purpose graphical processor units (GPUs) accelerated GMM computation but they do not present pruning solution. Bilmer [12] outlines an adaptive, on-line pruning approach which is very complex and is limited to 200 words. A WFST solution with a hardware accelerated Gaussian function and software search but with a new beam-pruning algorithm is presented in [13]; this is closest to our work as it uses adaptive pruning but they argue against histogram pruning due to the need to sort tokens which they argue is computationally complex; we counteract this by combining an online adaptive pruning with simple sorting strategy or histogram pruning which reduces the computation. The work in [14] also explores non-epsilon

---

**Fig. 11: Decision flowchart for decide insertion case.**

![Decision flowchart for decide insertion case.](image)

**Fig. 12: Representation of tokens sorted in sub-divisions of the beam width.**

**TABLE II: Word recognition accuracy with histogram pruning set at 4000 tokens.**

<table>
<thead>
<tr>
<th>Maximum number of tokens propagated (Memory needed (Mbit))</th>
<th>Unsor ted tokens</th>
<th>Sorted tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>30000 (345)</td>
<td>87.5%</td>
<td>88.3%</td>
</tr>
<tr>
<td>5000 (58)</td>
<td>85.5%</td>
<td>88.3%</td>
</tr>
<tr>
<td>4000* (46)</td>
<td>82.6%</td>
<td>88.2%</td>
</tr>
</tbody>
</table>

* corresponds to a bandwidth of 316Mbit/s
state removal by flattening the graph, but they dismiss it as it has poor performance due to its non-consecutive memory organization. For embedded systems, this does not cause the same problem and the ability to pre-cache the next frame overcomes any loss in throughput due to data divergence.

V. Conclusion

In this paper, an epsilon removal technique is presented which allows more efficient use of the memory by reducing memory accesses and enables loading of WFST data in advance, allowing application of look-ahead techniques. This allows pipelining of the different decoder tasks and allows the Gaussian cost to be accelerated. Following on from this, two pruning techniques were proposed; these save on the internal memory usage and number of operations processed.

References


Louis-Marie Aubert received a M.Eng in Electronic and Communication Systems and a M.Sc in Communication and Network in 2002 from the Institut National des Sciences Appliques (INSA), Rennes, France. He received a Ph.D. in 2005 from INSA for his work with Mitsubishi Electric on UWB communications. He has used his solid background in signal processing and algorithm to create innovative work in multiple domains from digital communication to speech recognition to video processing. He is co-author of 7 papers. Since 2011, he has worked for Application Solution (Electronic and Vision) Limited which specializes in embedded video processing in the automotive industry.

Roger Woods (SM’00) received a B.Sc degree (Hons.) in Electrical and Electronic Engineering and a PhD from the Queen’s University of Belfast in 1985 and 1990 respectively. He is a full professor at the same university and has created and leads the Programmable Systems Laboratory and his research interests are in heterogeneous programmable systems and system level design tools for a range of data, signal and image processing as well as telecommunications. He holds 4 patents, has published over 170 papers. Roger is a member of the IEEE Signal Processing and Industrial Electronics Societies and is on the Advisory Board for the IEEE SPS Technical Committee on the Design and Implementation of Signal Processing Systems. He has co-founded a company, Analytics Engines, which looks to exploit a lot of the programmable systems research.

Scott Fischaber (M’07) received a B.Sc. degree in Engineering with a specialty in Electrical Engineering from Colorado School of Mines, USA and a Pd.D. degree in memory architectures for FPGA-based heterogeneous systems from the Queen’s University, Belfast, UK, in 2003 and 2007, respectively. He holds two patents and has a number of publications in the area of optimisation and design of memory subsystems for a variety of application areas, including image, video, and speech processing. His research interests include FPGA-based hardware acceleration for high performance image and data processing, data analytics, and DSP. In 2008 he co-founded Analytics Engines Ltd. which specialises in high performance data analytics and accelerated computing; he currently holds the post of Chief Engineering Officer.

Richard Veitch received his BEng. degree in Electrical and Electronic Engineering from University of Aberdeen, UK in 2005. He has studied reconfigurable computing for various applications such as subsea digital holography, automated speech recognition and high performance computing. He currently works for Maxeler Technologies as a Field Applications Engineer.