Implementing a Hidden Markov Model Speech Recognition System in Programmable Logic

S.J. Melnikoff, S.F. Quigley & M.J. Russell
School of Electronic and Electrical Engineering, University of Birmingham, Edgbaston, Birmingham, B15 2TT, United Kingdom
s.j.melnikoff@iee.org

Presented at:
11th International Conference on Field Programmable Logic and Applications (FPL 2001)

Published in:
Lecture Notes in Computer Science #2147, pp.81-90

Implementing a Hidden Markov Model Speech Recognition System in Programmable Logic

S.J. Melnikoff, S.F. Quigley & M.J. Russell

School of Electronic and Electrical Engineering, University of Birmingham, Edgbaston, Birmingham, B15 2TT, United Kingdom
s.j.melnikoff@iee.org, s.f.quigley@bham.ac.uk, m.j.russell@bham.ac.uk

Abstract. Performing Viterbi decoding for continuous real-time speech recognition is a highly computationally-demanding task, but is one which can take good advantage of a parallel processing architecture. To this end, we describe a system which uses an FPGA for the decoding and a PC for pre- and post-processing, taking advantage of the properties of this kind of programmable logic device, specifically its ability to perform in parallel the large number of additions and comparisons required. We compare the performance of the FPGA decoder to a software equivalent, and discuss issues related to this implementation.

1 Introduction

The decoder part of a speech recognition system, i.e. the part that converts a pre-processed speech waveform into a sequence of words or sub-word units, is highly computationally demanding.

Current systems work best if they are allowed to adapt to a new speaker, the environment is quiet, and the user speaks relatively carefully; any deviation from this "ideal" will result in significantly increased errors. At present it is not clear whether these problems can be overcome by incremental development of the current algorithms, or whether more fundamental changes are needed. In either case, it is likely that the result will place increased computing demands on the host computer. Hence, as is the case for graphics, it may be advantageous to transfer speech processing to some form of co-processor or other hardware implementation.

Research has been carried out in the past on such implementations, generally using custom hardware. However, with ever more powerful programmable logic devices being available, these chips appear to offer an attractive alternative.

Accordingly, this paper describes the current findings of research into implementing the decoder part of a speech recognition system on a programmable logic device, targeting in particular an FPGA. We describe our most recent implementation, which follows on from the preliminary findings described in [1].

The paper is organised as follows. Section 2 explains the motivation behind the research; this is followed in section 3 by an overview of speech recognition theory and Hidden Markov Models. In section 4, we look at current commercial recognition ASICs, before describing the structure of the system in section 5, followed by details of the implementation and discussion of the results in section 6. Section 7 summarises the conclusions drawn so far, and section 8 describes issues that will need to be considered when working on future implementations.

2 Motivation

The increased computational power that a dedicated speech co-processor within a PC would provide could be utilised to improve the quality of the recognition. With more power at the user's disposal, it might be possible to make the system speaker-independent, and less susceptible to errors caused by a noisy environment (e.g. by using real-time noise compensation).

The results described below suggest that a hardware recogniser can perform speech-to-text transcription significantly faster than real time. At its simplest, this could be used for offline transcription, where a recording of speech could very quickly be converted to text.

Also, Viterbi decoding (described below) is often given as an example of a dynamic programming problem. Lessons learnt from a successful FPGA implementation of this could prove applicable to dynamic programming algorithms in non-speech applications.

It should be noted that the unique properties of speech mean that it only needs to be sampled at 100 Hz, giving us up to 10ms to perform all necessary processing. While this may beg the question as to whether we need to perform these calculations using dedicated hardware at all, we can take advantage of this free time to increase the accuracy of the system by improving the algorithm or speech model, so as to do in hardware what it would not be possible to do in software in the same amount of time.

3 Speech Recognition Theory

3.1 Overview

The most widespread and successful approach to speech recognition is based on the Hidden Markov Model (HMM) [2], [4], and is a probabilistic process which models spoken utterances as the outputs of finite state machines (FSMs). The notation here is based on [2].

3.2 The Speech Recognition Problem

The underlying problem is as follows. Given an observation sequence $O = O_0, O_1 \ldots O_{T-1}$, where each O_t is data representing speech which has been sampled at fixed intervals, and a number of potential models *M*, each of which is a representation of a particular spoken utterance (e.g. word or sub-word unit), we would like to find the model *M* which best describes the observation sequence, in the sense that the probability *P*(*M*|*O*) is maximised (i.e. the probability that *M* is the best model given *O*).

This value cannot be found directly, but can be computed via Bayes' Theorem [4] by maximising *P*(*O*|*M*). The resulting recognised utterance is the one represented by the model that is most likely to have produced *O*. The models themselves are based on HMMs.

3.3 The Hidden Markov Model

An *N*-state Markov Model is completely defined by a set of *N* states forming a finite state machine, and an $N \times N$ stochastic matrix defining transitions between states, whose elements a_{ij} = *P*(state *j* at time *t* | state *i* at time *t*-1); these are the *transition probabilities.*

With a Hidden Markov Model, each state additionally has associated with it a probability density function $b_j(O_t)$ which determines the probability that state *j* emits a particular observation O_t at time *t* (the model is "hidden" because any state could have emitted the current observation). The p.d.f. can be continuous or discrete; accordingly the pre-processed speech data can be a multi-dimensional vector or a single quantised value. $b_j(O_t)$ is known as the *observation probability.*

Such a model can only generate an observation sequence $O = O_0, O_1 \ldots O_{T-1}$ via a state sequence of length *T*, as a state only emits one observation at each time *t*. The set of all such state sequences can be represented as routes through the state-time trellis shown in Fig. 1. The $(j,t)^{\text{th}}$ node (a state within the trellis) corresponds to the hypothesis that observation O_t was generated by state *j*. Two nodes (*i*,*t*-1) and (*j*,*t*) are connected if and only if $a_{ij} > 0$.

As described above, we compute *P*(*M*|*O*) by first computing *P*(*O*|*M*). Given a state sequence $Q = q_0, q_1 \ldots q_{T-1}$, where the state at time *t* is q_t, the joint probability, given a model *M*, of state sequence *Q* and observation sequence *O* is given by:

$$P(O,Q|M) = b_0(O_0) \prod_{t=1}^{T-1} a_{q_{t-1}q_t} b_{q_t}(O_t), \tag{1}$$

assuming the HMM is in state 0 at time *t* = 0. *P*(*O*|*M*) is then the sum of all possible routes through the trellis, i.e.

$$P(O|M) = \sum_{\text{all } Q} P(O,Q|M). \tag{2}$$

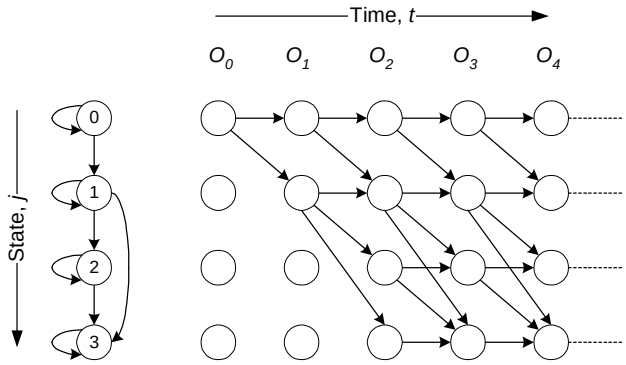

Fig. 1. Hidden Markov Model, showing the finite state machine for the HMM (*left*), the observation sequence (*top*), and all the possible routes through the trellis (*arrowed lines*)

3.4 Viterbi Decoding

In practice, the probability $P(O|M)$ is approximated by the probability associated with the state sequence which *maximises* $P(O,Q|M)$. This probability is computed efficiently using Viterbi decoding.

Firstly, we define the value $\delta_t(j)$, which is the maximum probability that the HMM is in state j at time t. It is equal to the probability of the most likely partial state sequence $q_0, q_1 \ldots q_t$, which emits observation sequence $O = O_0, O_1 \ldots O_t$, and which ends in state j:

$$\delta_t(j) = \max_{q_0, q_1 \ldots q_t} P(q_0, q_1 \ldots q_t; q_t = j; O_0, O_1 \ldots O_t | M). \tag{3}$$

It follows from equations (1) and (3) that the value of $\delta_t(j)$ can be computed recursively as follows:

$$\delta_t(j) = \max_{0 \le i \le N-1} [\delta_{t-1}(i) a_{ij}] \cdot b_j(O_t), \tag{4}$$

where i is the previous state (i.e. at time t-1).

This value determines the most likely predecessor state $\psi_t(j)$, for the current state j at time t, given by:

$$\psi_t(j) = \arg\max_{0 \le i \le N-1} [\delta_{t-1}(i) a_{ij}]. \tag{5}$$

At the end of the observation sequence, we backtrack through the most likely predecessor states in order to find the most likely state sequence. Each utterance has an HMM representing it, and so this sequence not only describes the most likely route through a particular HMM, but by concatenation provides the most likely sequence of HMMs, and hence the most likely sequence of words or sub-word units uttered.

4 Speech Recognition ASICs and Cores

In order to better appreciate how this implementation compares to existing commercial speech recognition hardware and firmware, a few examples are given below.

4.1 ASICs

Sensory RSC-300 & RSC-364. These two chips [6] are based on an 8-bit 14MHz RISC microprocessor, and use a pre-trained neural network to perform speaker-independent speech recognition. They can store 6 speaker-dependent words in on-chip RAM, with the RSC-300 using off-chip storage to enable no limit on vocabulary size (subject, of course, to access times). They boast speaker-independent recognition accuracy of 97%, with 99% for speaker-dependent, and a response time of 83ms.

These devices are designed for use in "consumer electronics products," with similar chips targeted for use in toys.

Sensory Voice Direct 364. This device is also based on a neural network, but is purely speaker-dependent. It can operate under the control of a separate microprocessor, permitting a 60-word vocabulary, or on its own which limits it to 15 words. It is capable of 99% accuracy, with a response time of less than 500ms.

Philips HelloIC. This device [5] uses a 16-bit fixed point DSP to allow both speaker-dependent and independent speech recognition, with a vocabulary of 15 words. It operates at 30MHz, and is capable of "noise robustness" and echo cancellation.

DSPs. Some DSPs have dedicated logic for Viterbi decoding. For example, the Texas Instruments TMS320C54x family [7] has a Compare, Select, and Store Unit which is used for the add-compare-select part of the decoding process. As with comparable FPGA cores, however, this is designed more for signal processing applications than speech recognition.

4.2 FPGA Cores

A comparable FPGA core is TILAB's Viterbi Decoder core [8]. On a Virtex XCV50-6, it requires 495 slices, and runs at 56MHz - but only accepts 6-bit inputs. This is because Viterbi decoding was originally designed for signal processing applications, specifically for the decoding of convolutional codes; hence the core is best suited to that kind of application, and not speech, which tends to require larger data widths.

Xilinx also has a decoder core [9], again designed for signal processing rather than speech recognition, which provides a choice between a parallel version, requiring 1000-2000 Virtex-II slices running at 90-130 MHz, and a serial one, requiring around 500 slices operating at 200 MHz.

5 System Design

5.1 System Structure

The complete system consists of a PC, and an FPGA on a development board inside it. For this implementation, we used already pre-processed speech waveforms.

The waveforms are quantised, and the resulting data sent to the FPGA, which performs the decoding, outputting the set of most likely predecessor states. This is sent back to the PC, which performs the simple backtracking process in software.

5.2 FPGA Implementation Structure

The structure of the implemented design is shown in Fig. 2. The HMM Block contains the processing elements which compute $\delta_t(j)$, the probability of the observation sequence up to time t and the "best" partial state sequence terminating in state j at time t; and the most probable predecessors $\psi_t(j)$. The $\delta_t(j)$ values are then passed to the Scaler, which scales the data in order to reduce the required precision, and discards values which have caused an underflow.

In order to keep the design as simple as possible, no language model is being used. As a result, the probability of a transition from one HMM's exit state to another's entry state is the same for all HMMs. This value is computed by a dedicated block, which passes it to Delta Delay. Delta Delay is used at initialisation to set the values of $\delta_0(j)$, and thereafter routes the scaled values of $\delta_t(j)$ back to the HMM Block, while ensuring that the various data streams are properly synchronised.

The node is the basic processing unit for performing Viterbi decoding, and each one processes the data corresponding to one state of an HMM. As every node depends only on the outputs of nodes produced in the previous time frame, and not the current one, the nodes can all be implemented in parallel.

The structure of the nodes is very similar to that used in previous parallel implementations, and is shown in Fig. 3. All calculations are performed in the log domain. This makes the system particularly appropriate for FPGAs, as it reduces all

the arithmetic to addition-type operations (in this case, addition, comparison and subtraction), for which FPGAs tend to have dedicated logic.

The most likely predecessor $\psi_t(j)$ information is stored off-chip, and processed in software, as it requires more storage than is available on the FPGA, and does not demand much processing power.

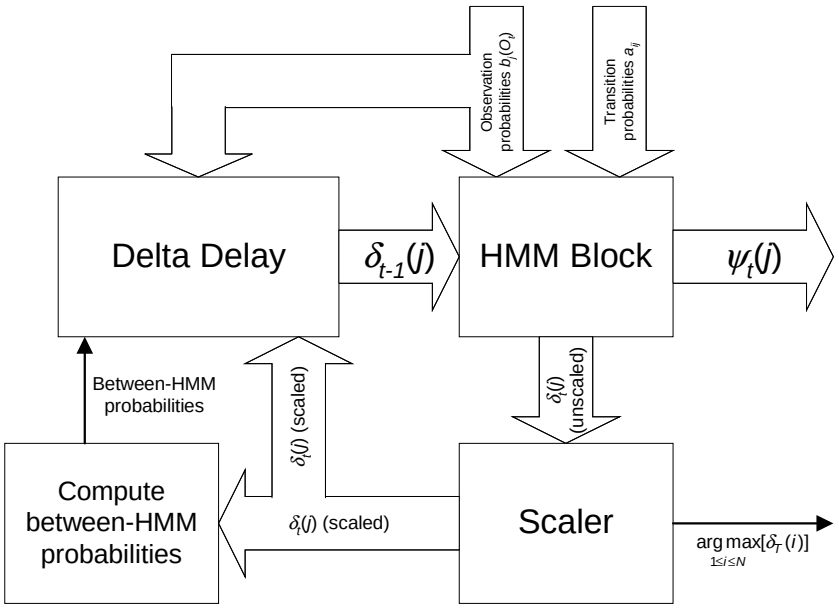

Fig. 2. System structure. The HMM Block contains the processing elements - nodes - which compute $\delta_t(j)$, the probability of the observation sequence up to time t and the "best" partial state sequence terminating in state j at time t; and the most probable predecessors $\psi_t(j)$

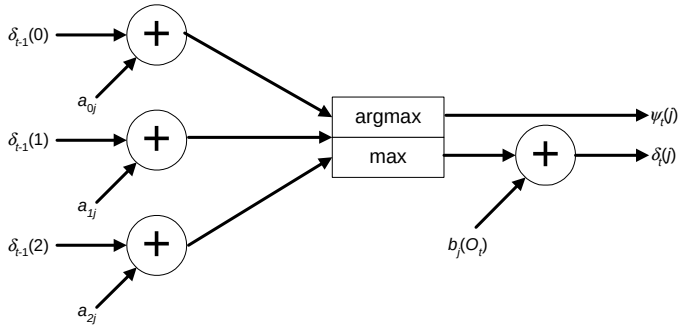

Fig. 3. Node structure. Each node processes the data corresponding to one state of an HMM. As every node depends only on the outputs of nodes produced in the previous time frame, and not the current one, the nodes can all be implemented in parallel

5.3 Data Format

For this implementation, we used a simple model consisting of 49 monophones - i.e. just vowels and consonants - of 3 states each, with no language model, which gave a recognition accuracy of 50%. The data for this model was calculated using the HTK speech recognition toolkit [3], based on a set of recorded speech waveforms from the TIMIT database. The speech observations themselves consisted of 8-bit values, treated as addresses into 256-entry 15-bit-wide look-up tables, one for each node.

6 Implementation and Results

The design described above was implemented on a Xilinx Virtex XCV1000, sitting on Celoxica's RC1000-PP development board. The RC1000 is a PCI card, which features 8Mb of RAM, accessible by both the PC and FPGA, 8-bit data ports in each direction with handshaking control, and two 1-bit data lines.

The RC1000 was used within a PC with a Pentium III 450MHz processor. C++ software was written to do pre- and post-processing, and was also capable of carrying out all the same calculations as the FPGA, in order to compare performance. The code was written so as to be as functionally similar to the VHDL as possible, with the exception that scaling was found to be unnecessary.

6.1 Implementation Overview

The design was implemented in three versions, each able to process a different number of HMMs in parallel. The first attempted to process all 49 HMMs in parallel, but did not fit into the XCV1000, requiring 100% of the FPGA's slices, which along with a $\delta_t(j)$ data bus over 2000 bits wide resulted in a design which could not be routed.

In order to substantially reduce the required resources, a second version was implemented which cut the number of parallel HMMs to 7, with the other modules in the system reduced in size accordingly. The final design required 70% of the slices, and routed successfully.

However, this implementation was still not satisfactory, as there was a significant bottleneck when it came to reading from and writing to off-chip RAM. In addition, any future implementation based on a more complex algorithm (e.g. continuous HMMs) would have no more space on the FPGA with which to perform additional calculations as required.

In order to overcome these problems, a third version was designed which dealt with just one HMM at a time, reducing the resource usage, and matching the internal bandwidth with that available for accessing the RAM, thereby reducing the delay incurred by the second implementation.

6.2 Data Storage

We previously [1] compared the use of off-chip RAM, on-chip distributed RAM and on-chip Block RAM, and in this implementation, we use all three. The observation probabilities form the largest block of data - nearly 70Kb. This amount is too large to store on the FPGA, and in any case we do not need it all at once, so the data is stored in the RAM banks on the RC1000 board, and the relevant parts loaded in when a new observation is received.

The transition probabilities are somewhat smaller at 830 bytes, and so are stored in Block RAM. The between-HMM probabilities occupy even less space, and are stored in distributed RAM.

6.3 Hardware

7-HMM Implementation. The implementation tools (Xilinx Foundation Design Manager 2.1i SP6) reported a maximum clock frequency of 31MHz. The design required 26 cycles to process a single observation, but because the internal data bus was larger than the off-chip RAM data bus, further delays were incurred: 36 cycles to read in the observation probabilities ($b_j(O_t)$)from RAM and 20 to write the predecessor information ($\psi_t(j)$)back to RAM. As some of the RAM accesses could take place while data was being processed, a total of 77 cycles were required.

Hence from these figures, we expected a complete observation cycle to take around 2.7μs. Experiments showed the average time (taken to be the mean time to process each observation value and write the results to RAM) to be 2.1μs.

1-HMM Implementation. This design returned a maximum clock frequency of 86.4MHz. This higher value is partly due to greater use of pipelining and relationally-placed macros, and gives a predicted time per observation of 2.5μs.

The narrower data width led to a longer pipeline - 117 cycles. However, the effective delay due to RAM accesses was reduced, as data from RAM could enter the pipeline as soon as it was read, rather than being buffered first as was done in the previous implementation.

6.4 Comparison with Software

The equivalent processing was performed in software on the same PC as used above. The average time per observation (taken as the time required to compute the most likely predecessor information otherwise done by the HMM Block, and the between-HMM probabilities) was found to be 790μs.

For the 7-HMM implementation, this gives a speedup over software of 370, and we predict that the 1-HMM implementation will give a speedup of at least 310.

7 Conclusion

We have implemented a simple monophone HMM-based speech recogniser, using an FPGA to perform the decoding, and a host PC to send it the speech observations, and process the results. We have also implemented the decoder in software in order to compare performance.

Our findings so far are that while the FPGA can easily outperform the PC, the degree to which the algorithm can be parallelised within the FPGA is limited by the bandwidth restriction between the FPGA and RAM, leading to a less parallel implementation than originally planned. However, this implementation is expected to operate at a comparable speed to its predecessor, while leaving more space free on the device for use in the future.

8 Further Issues

- **Types of HMM:** We mention above that the speech data is quantised; such discrete HMM-based systems tend to perform poorly in term of accuracy. A better approach is to use continuous HMMs, where the data is processed to extract feature vectors, with the observation probability distribution based on Gaussian mixtures.
- **Real-time recognition:** This system is essentially an off-line transcription system, in that the speech data is sent to the FPGA and processed as fast as is possible, and backtracking does not take place until the (known) end of the speech. Since, for a real-time continuous speech recognition system, the length of the speech data is not known in advance, other methods exist for initiating backtracking. The pre- and post-processing can be done in software in real time.

References

1. Melnikoff, S.J., James-Roxby, P.B., Quigley, S.F. & Russell, M.J., "Reconfigurable computing for speech recognition: preliminary findings," *FPL 2000, LNCS #1896*, pp.495-504.
2. Rabiner, L.R., "A tutorial on Hidden Markov Models and selected applications in speech recognition," *Proceedings of the IEE*E, **77**, No.2, 1989, pp.257-286.
3. Woodland, P.C., Odell, J.J., Valtchev, V. & Young, S.J. "Large vocabulary continuous speech recognition using HTK," *ICASSP '94*, **2**, pp.125-128.
4. Young, S., "A review of large-vocabulary continuous-speech recognition," *IEEE Signal Processing Magazine*, **13**, No.5, 1996, pp.45-57.
5. http://www.speech.philips.com/ud/get/Pages/vc_home.htm
6. http://www.sensoryinc.com/
7. http://dspvillage.ti.com/docs/dspproducthome.jhtml
8. http://www.xilinx.com/ipcenter/catalog/search/alliancecore/tilab_viterbi_decoder.htm
9. http://www.xilinx.com/ipcenter/catalog/search/logicore/viterbi_decoder.htm