

# Hardware Speech Recognition in Low Cost, Low Power Devices

Sukun Kim, Sergiu Nedevschi and Rabin K Patra  
Computer Science Division (University of California, Berkeley)  
CS252 Class Project, Spring 2003  
{binetude, sergiu, rkpatra}@cs.berkeley.edu

**Abstract**— We investigate the problem of providing speech recognition functionality, as an alternative to visual user-interfaces, in hardware devices facing important power, cost and storage constraints. Our goal is to enable these devices to perform continuous real-time hardware speech recognition for languages with small vocabularies, up to tens of words, and having limited grammatical constructions. Real-time continuous speech recognition is a computationally-intensive, but highly parallelizable task, with most of the computation taking place in the decoding stage of speech recognition. To this end, we propose a custom hardware implementation of the decoder part of a HMM-based speech recognition system, minimizing cost and power consumption while maintaining a good recognition accuracy. We present our preliminary findings, using a simulator to evaluate our design.

## I. INTRODUCTION AND RELATED WORK

Designing low-cost, flexible and customizable user-interfaces is a key component in providing an infrastructure for information technology access for developing regions. Such user-interfaces would be running on hand-held devices that may have severe power, cost and size constraints.

As we cannot assume that the users of such a system will be literate, spoken language input and output plays a major part in the design of user-interfaces. However, reliable recognition of large-vocabulary fluent speech is still not state of the art, but providing cheap, reliable and highly accurate speech recognition features for isolated word or limited vocabulary systems should be achievable. Providing inexpensive speech recognition will also alleviate the need for costly LCD interfaces in these systems. Given the constraints for obtaining expert support in developing regions, our objective should be a system that can be deployed with a base training (speaker-independent models) with update and training facilities. The system should also be flexible enough to handle different dialects and speech model parameters with minimal effort to change dialects, ideally requiring only a download of the new model. While there are current speech recognition toolkits (HTK [1]) that can provide good accuracy with limited vocabulary speech, they are too heavyweight to be deployed on a low power and low cost device. Implementing speech recognition on a general purpose CPU is quite computation intensive.

However, it is known that significant power and cost

savings can be obtained if custom designs (ASIC) are made for specific purposes. Especially with speech recognition, a major part of the computation that is digital signal processing can be implemented in custom hardware. What we explore in this paper is the possibility doing the other component of recognition that is decoding on the recognition network also in hardware. While the problem of implementing speech recognition on hardware has been approached ([2]) and many commercial products ([3] [4] [5] [6]) are available nowadays, most of the solutions are targeted specifically to a particular language (like English) and are not designed to be extensible to handle different dialect models or vocabularies of different sizes. The other factors going against them are cost ([5] needs a TI DSP chip, [2] needs FPGAs) and therefore, they are not suitable for low cost handheld devices. For a more detailed list of hardware speech recognition solutions, look at [7].

Decoding for speech recognition basically involves solving a dynamic programming problem in time. In this paper we show that recognition for a limited vocabulary language (tens of words) can be parallelized using a small number of very simple processing elements with limited memory and clock speed requirements. The whole recognition process consists of two steps - in the first step, these processing elements are programmed with the language grammar and the trained model parameters by a general purpose CPU. The second step involves the actual recognition performed by these processing elements using coded speech vectors from a custom ASIC and requires no supervision from the general purpose CPU. The architecture proposed by us would also be able to handle different speech encoding algorithms, and different language models. Similar proposals for hardware accelerators aiding in speech recognition has been proposed before ([8]).

We use the HTK (Hidden Markov Toolkit) ([9]) from the Cambridge University Engineering Department for our development and testing environment. For evaluating our design, we perform a workload style analysis using coarse simulation of the proposed hardware.

### A. Motivation

The key motivations that were driving us:

- **Low Power/Cost:** As our solution was to be targeted toward low cost handhelds, we had to choose a design that would run at low clock speeds and voltages. This meant that we could not do full-fledged recognition on a general purpose CPU, but a limited vocabulary based user-interface system was adequate. Also a key power reduction technique is to reduce the voltage at which a circuit runs by parallelizing the design([10]), and therefore we wanted a design that would be easily divided into parallel operations.
- **Flexibility:** As we wanted our solution to work with a range of language dialects and models, we wanted a system that could easily download a new grammar and pre-trained set of parameters.
- **Scalability:** We did not want our system to be restricted to a fixed sized vocabulary or a particular speech coding algorithm, therefore a clean and scalable design was needed that could handle more processing elements without much parallelization overhead.

These issues are discussed in more detail in section III.

## B. Overview

In section II, we present the mathematical theory behind HMM based speech recognition and the structure of a typical recognition system. Section III, presents the design issues that guided us and in section IV, we discuss the proposed architecture for our solution. Section VI deals with the experimental evaluation and results.

## II. SPEECH RECOGNITION BASICS

The chapter outlines the main components of a classic speech recognition system and their functioning. We limit ourselves to presenting the notions necessary for understanding the context of our work. For more in-depth coverage of the subject, please see [11].

Figure 1 shows a HMM-based speech recognizer broken down into three main stages. The spectral analysis stage, also called the feature extraction stage, is responsible for dividing the waveform inputs into frames (typically 25 milliseconds in length), that are then transformed into spectral *feature vectors*. These vectors convey information about the energy of the speech signal for each of its component frequencies. These vectors are then fed into a decoder, that computes a set of most probable sequences of words given the acoustic events observed. The decoder uses a recognition network on which it performs dynamic programming in order to compute the sequences of words having the highest probability. The recognition network is constructed using a lexicon of phonemes, represented by Hidden Markov Models (HMMs), a word dictionary represented again by HMMs, and a *n-gram* grammar evaluating the probability of a word considering the previous *n* word occurrences. Finally, syntactic and semantic analysis is used to select among the set of most probable sequences.

### A. Spectral Analysis

This component of a speech recognition system that is the signal-processing front-end of a recognizer is common to almost all recognition systems. The basic aim of spectral analysis is to represent the acoustic events in a speech signal in terms of an efficient set of speech parameters. The two most common choices for the signal-processing front-end are a bank-of-filters and an LPC model.

- **Bank-of-filters:** In this model, the speech signal is passed through a bank of *Q* bandpass filters whose coverage spans the frequency range of the speech signal(100Hz to 3000 Hz). The individual filters do not overlap in frequency and the output of the *i*th bandpass filter is centered around  $\omega_i$ . Since, the human ear has non-linear response, the different bands might be of varying widths.
- **Linear Predictive Coding(LPC):** This model provides a set of LPC parameters that specify the spectrum of an all-pole model that best matches the spectrum of the speech frame. The LPC model is preferred sometimes as it provides a good representation of the human voice signal, and it is more analytically tractable to implement either in software or hardware.
- **Vector Quantization:** Though the filter-bank and LPC model based speech coding methods reduce the information rate of the speech signal by a factor of around 10, in more constrained requirements, we may use just a single spectral representation of the basic speech unit. This is the idea behind Vector Quantization(VQ) methods, where a codebook of unique speech vectors is used to quantize and map a speech frame to an entry of a precomputed codebook. This has the capability to reduce the vector representation size even further and also involves lesser computation later on in the decoding stage as the probability calculation is reduced to just a table lookup.

### B. Decoding

The problem the decoder has to solve is the following: given an observation sequence  $O = O_1, O_2 \dots O_n$ , where each  $O_i$  represents a feature vector, and a set of models  $M$ , each being a representation for a spoken utterance, the decoder tries to find the model that best matches the observation sequence, such that the probability  $P(O|M)$  is maximized. This probability is not directly computable, but can be estimated by using the Bayes theorem and computing  $P(M|O)$ . The resulting recognized utterance is the one represented by the model having the maximum probability among all models.

The models are represented using Hidden Markov Models (HMMs). An *N*-state Markov Model is defined by a set of *N* states forming a finite state machine, and an  $N \times N$  matrix whose elements  $a_{ij}$  define the transition probabilities between states *i* and *j*.

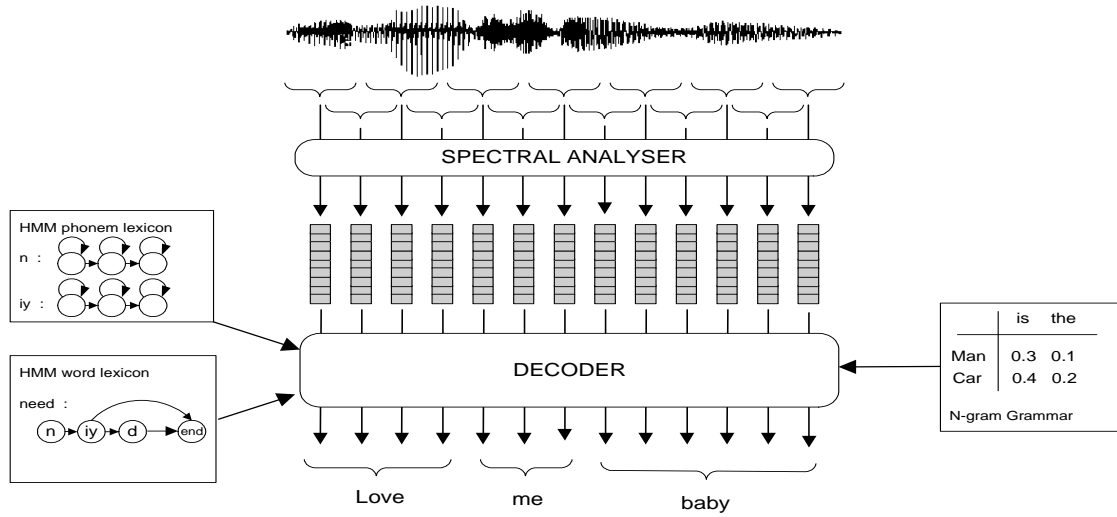


Fig. 1. Structure of a speech recognition system

In the Hidden Markov Model, each state is additionally associated with a probability density function  $b_j(O_t)$  representing the probability that a particular observation  $O$  is emitted by state  $j$  for observation number  $t$ . The probability distribution can be continuous, in which case the speech data is a multi-dimensional vector, or discrete, corresponding to single quantized values of the data.  $b_j(O_t)$  is known as the observation probability. Both the transition probabilities and the observation probability densities for each of the states of an HMM modeling a spoken utterance are estimated during a training process.

For the discrete case, the output probability for each state is specified by a lookup table, describing the probabilities for each of the possible discrete values. However, for the continuous density case, the observation distributions are represented by Gaussian Mixture Densities. The formula for  $b_j(O_t)$  is:

$$b_j(O_t) = \prod_{s=1}^S \left[ \sum_{m=1}^{M_s} c_{j s m} N(O_{st}; \mu_{j s m}, \Sigma_{j s m}) \right]^{\gamma_s}, \quad (1)$$

where  $M_s$  is the number of mixture components in stream  $s$ ,  $c_{j s m}$  is the weight of the  $m^{th}$  component and  $N(O; \mu, \Sigma)$  is a multivariate Gaussian with mean vector  $\mu$  and covariance matrix  $\Sigma$ :

$$N(O; \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} e^{-\frac{1}{2}(O-\mu)' \Sigma^{-1}(O-\mu)} \quad (2)$$

where  $n$  is the dimension of the observation vector  $O$ .

The computation of  $P(O|M)$  is described in the following. Each sequence of observations  $O_1 O_2 \dots O_n$  is emitted by a state sequence of length  $n$ . Such a sequence is described by a path in the state network. Given a state

sequence  $Q = q_1 q_2 \dots q_n$ , where the state at time  $t$  is  $q_t$ , the joint probability of the state sequence  $Q$  and an observation sequence  $O$  given a model  $M$ , can be expressed as follows:

$$P(O, Q|M) = b_1(O_1) \prod_{t=2}^n a_{q_{t-1} q_t} b_{q_t}(O_t) \quad (3)$$

assuming that state 1 is the start state of the network. The probability  $P(O|M)$  is the sum of the probabilities corresponding to all the possible paths through the network of states:

$$P(O|M) = \sum_{\text{all } q} P(O, Q|M) \quad (4)$$

In practice, the probability  $P(O|M)$  is approximated by the probability of the state sequence maximizing  $P(O, Q|M)$ . This probability can be easily estimated using Viterbi decoding.

For a given model  $M$ , let  $\psi_j(t)$  represent the maximum likelihood of having observed the sequence  $O$ , and being in state  $j$  at time  $t$ . This partial likelihood can be computed efficiently using the following recursion:

$$\psi_j(t) = \max_i \{ \psi_i(t-1) a_{ij} \} b_j(O_t), \quad (5)$$

given that

$$\psi_1(1) = 1, \text{ and } \psi_j(1) = a_{1j} b_j(O_1). \quad (6)$$

for  $1 < j < N$ , being the total number of states of the model. The maximum likelihood  $P_m(O|M)$  is then given by:

$$\psi_N(n) = \max_i \{ \psi_i(n) a_{iN} \}. \quad (7)$$

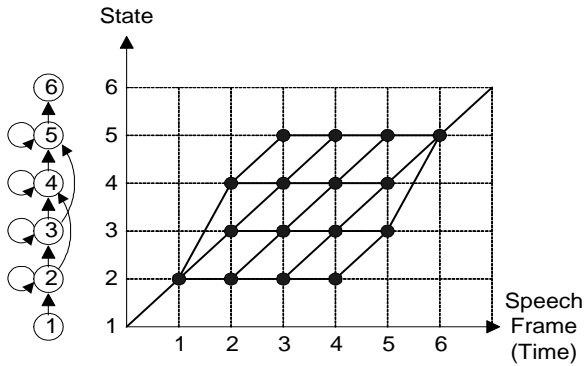


Fig. 2. Viterbi Algorithm

Since the computation supposes repeated multiplications with small values, it leads to underflow, and, in order to prevent this, log probabilities are used instead. Hence, the recursion from equation 5 becomes:

$$\psi_j(t) = \max_i \{ \psi_i(t-1) + \log(a_{ij}) \} + \log(b_j(O_t)). \quad (8)$$

As illustrated in figure 2, the functioning of the algorithm can be seen as the process of finding the best path through a matrix, where the X-axis represent the speech frame number (time), and the Y-axis represents the state number. The cost of an edge represents the probability of the corresponding transition, and the cost for each node represents the observation probability at the corresponding state in the given time moment. We can easily notice that the problem can be solved using the dynamic probability principle, since the partial probability corresponding to each node must be computed only once.

The spoken utterances modeled by HMMs can be sub-word constructions such as monophones and triphones, or whole words. Moreover, words can be represented as Markov chains or HMMs of phonemes.

These models representing words are then aggregated using a language model. There are several possibilities in modeling the language, depending on its complexity. One of the most used models is the  $n$ -gram grammar [9], assigning probabilities to words as a function of the previous  $n$  words. However, for small and limited languages, simpler models such as context-dependent grammars or even regular grammars can be used.

### III. DESIGN ISSUES FOR OUR SYSTEM

There are several features that differentiate our recognition system from the general case. Our hardware recognizer is intended for use in the context of speech-triggered user interfaces. This drastically limits both the number of words supported in the vocabulary, as well as the complexity of the grammar describing the language. The system will need to support small languages of up to a few hundred words,

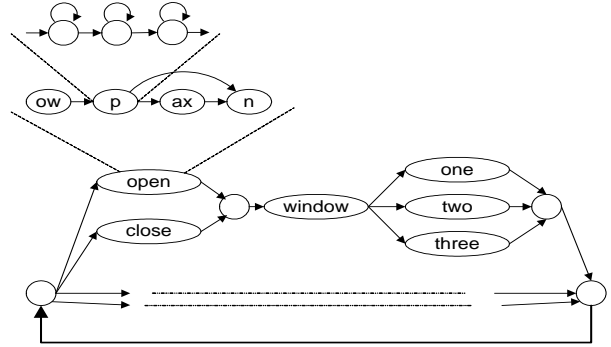


Fig. 3. Network expansion

and small sets of possible syntactic constructions, usually application-driving commands.

On the other hand, our system must be supported on a hardware platform subject to imperative constraints in terms of power consumption, storage capacity and fabrication cost. Moreover, we must consider the additional constraint of performing real-time recognition. All these limitations recommend the approach of implementing the recognition function on a custom-designed chip, and this can yield both smaller power consumption and better performance. This is especially true in the case of the decoder module, responsible for the most costly part of the computation, because the algorithms used to estimate the sequence probabilities are inherently parallel, and consequently custom parallel hardware can be used.

The above-mentioned considerations led us to make the following adaptations to the general recognition mechanism:

#### A. Word Model

As we know, the decoder uses HMMs to model individual phonemes, as well as individual words. The most efficient algorithm used to estimate the probabilities for these models is the Viterbi algorithm, which relies on the dynamic programming principle. This makes the probability computation both efficient and parallelizable.

This is unfortunately not the case with computing probabilities of sequences of words, if an  $n$ -gram language model is used, because the same dynamic programming principle does not apply in this case. Instead search algorithms such as A\* are used instead that adds a great deal of complexity to the computation, and limits the parallelization opportunities, making it expensive and impractical to implement in hardware. On the other hand, the benefits of using an  $n$ -gram model for small languages with restricted grammar are small.

Instead, we believe using regular grammars to constrain the set of possible word sequences would be enough to perform syntactic analysis in the decoding process. By doing this, we provide sufficient context information to

eliminate ambiguities among different words having very similar pronunciations. This solution, even though not viable for every language, proves to be perfectly suited to our case, where user-interface constructions can be easily expressed using regular expressions.

### B. Network Expansion

Having this language representation enables us to expand the recognition information given by the phoneme and word HMM models into a single recognition network, on which dynamic programming techniques can be used to compute the most probable path. An example of such a recognition network is illustrated in figure 3.

### C. Decoding Algorithm

The decoder uses a Viterbi-like algorithm, called the token-passing algorithm [12], to compute the path through the recognition network having the highest probability. The classic Viterbi algorithm computes the probability of the most probable path, without actually retaining the path. While this approach is enough for the case of isolated word recognition, where the sequence of sub-word states is usually uninteresting, it is not enough for larger networks, where we are interested in the sequence of words spanned by the winning path.

The token-passing algorithm makes the concept of state path explicit. Suppose each state  $i$  of an HMM at time  $t$  holds a single movable token that, among other information, contains the partial probability  $\psi_i(t)$ . The path extension algorithm represented by equation 8 is replaced by the following steps executed at each time frame  $t$ :

- 1) Pass a copy of the token in state  $i$  to all outgoing connecting states  $j$ , incrementing the log probability of the copy by  $\log[a_{ij}]$  (transition probability).
- 2) Examine the tokens in every state and discard all but the one with the highest probability.
- 3) Increment the log probability of the token by  $\log[b_j(o(t))]$ . (observation probability).

Please note that the above-presented algorithm assumes each state to be an emitting one.

In order for the algorithm to keep track of the history of a token's route, we need a set of history records, and every token must carry a pointer to one of these records. When a token is propagated from the exit state of a word to the entry state of another, the transition represents a potential word boundary. A new history record is created, containing the value of the new word, and a reference to the old history record the token was pointing to. Finally, the token is modified to point to this newly created record. Figure 4 shows details about how this algorithm works. At the end of the recognition process, the token emerging from the final state of the network will refer a history record, that can be traced back to obtain the full sequence of words the final token has passed through.

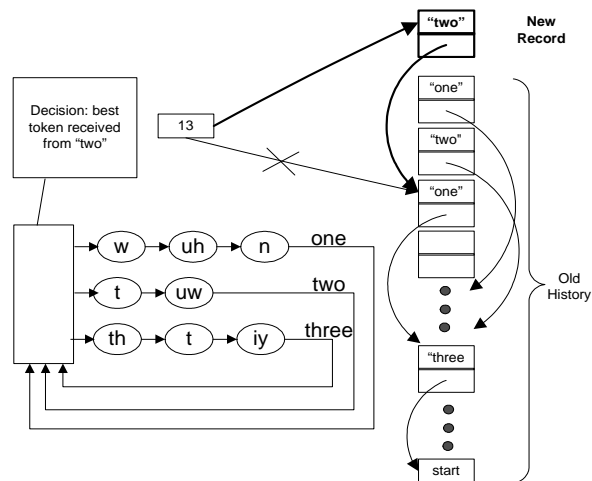


Fig. 4. History for a recognition network looking at sequences of the words "one", "two", "three". In the step shown here, the winning token among word end nodes is token number 13, coming from the word "two". A new history record is added, pointing to the previous record referenced by 13. Token 13, entering in parallel each of the three possible words, will point to the newly created record. This attests that token 13 has recognized the sequence: start, "three", "one", "two" so far. If 13 will be the overall winning token at the end of the recognition process, this sequence will be the beginning of the output sequence for the decoder (in reverse order)

## IV. PROPOSED ARCHITECTURE FOR THE DECODER

Using the token passing algorithm, the probability computation is distributed throughout the recognition network. This means that each node in the network is responsible for incrementing the probability of all the tokens it receives and then for choosing token with the maximum partial probability, while discarding the rest. The operations performed by all the nodes in the model are very similar, and almost all computations pertaining to one stage can be performed in parallel. As one can easily notice, the computation is very symmetric and inherently parallelizable, and a hardware design taking advantage of these properties can be easily imagined.

In order to establish a logical way of parallelizing the computation, we must look closer at the topology of the recognition network, and find out the similarities and differences of the computation performed at each node. Let's illustrate our reasoning using a simple example. Suppose our recognizer can receive as inputs any sequence of the digits *one*, *two* and *three*. In this case, we can use the recognition network shown in figure 5.

Unfortunately, as we can see, using this kind of network comes with some drawbacks:

- 1) The in-degree of the states at the beginning of each word is potentially unbound, depending on the grammar, and
- 2) A lot of token comparisons are redundant, since the all the outputs from the final word states are compared against each other at each of the input states.

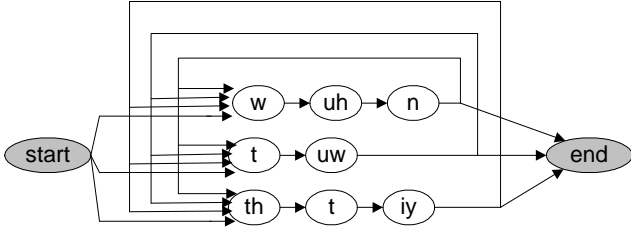


Fig. 5. Simple network for three words

The complexity of the hardware components performing the computation at the level of the network's nodes, as well as the data traffic between nodes, will greatly increase if the in-degree of the states is unbound. In order to alleviate the problem, we introduce dummy nodes that serve to aggregate data, and also reduce the number of edges in the network.

Returning to our example, the decoder's topology featuring the above-mentioned dummy nodes is presented in figure 6.

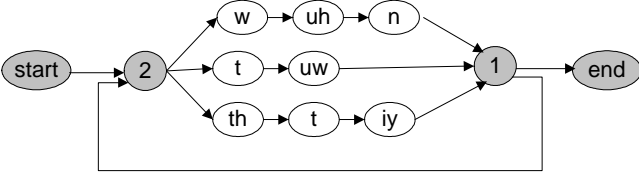


Fig. 6. Network of three nodes with dummy nodes

Now we can expand any regular grammar into a recognition network using Thompson's construction. Having done this expansion, the nodes of the network will have the following properties:

- 1) The non-dummy nodes have a bounded in-degree, determined by the topology of the HMMs used to model phonemes and words. Usually, the in-degree will be 2 for most states, rarely 3.
- 2) The dummy aggregating nodes have an unbound in-degree, depending on the size and structure of the grammar.
- 3) The non-dummy nodes are emitting i.e have non-zero observation probabilities, while dummy nodes are non-emitting.

#### A. Computation in the System

There are two kinds of computations, corresponding to regular nodes and aggregator nodes. At each time step, each node pulls its operands from the other nodes, and computes its own token value.

- Regular nodes: An example is node  $N2$  in figure 7. Using the token values at time  $t$ , node  $N2$  will compute the token value for the  $t + 1$  time step, by choosing the maximum between the values received from  $N1$  and itself. That is to say,

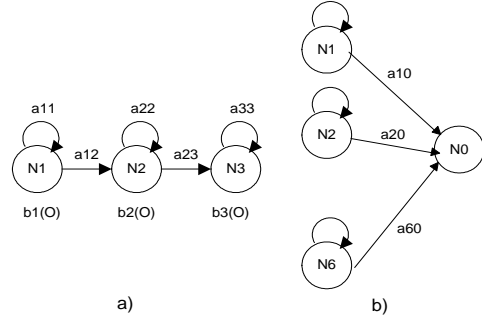


Fig. 7. Types of nodes/computation

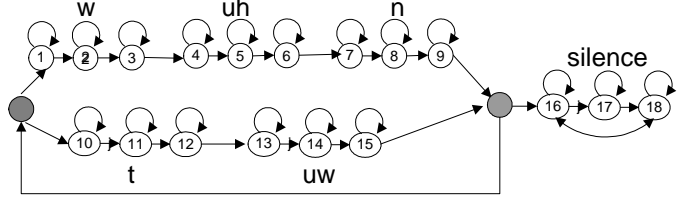


Fig. 8. Fully expanded network

$$\psi_2(t+1) = \max\{\psi_1(t) * a_{12} * b_2(O_{t+1}), \psi_2(t) * a_{22} * b_2(O_{t+1})\}$$

Since the observation probability,  $b_2(O_{t+1})$ , is the same for both terms, only  $\psi_1(t) * a_{12}$  and  $\psi_2(t) * a_{22}$  need to be compared. Moreover, log probabilities can be used. Consequently,  $\log(\psi_1(t)) + \log(a_{12})$  and  $\log(\psi_2(t)) + \log(a_{22})$  are evaluated and compared, and finally  $\log(b_2(O_{t+1}))$  is added.

The logarithm of the transition probabilities  $a_{12}$  and  $a_{22}$  can be pre-computed, and used for each computation. The evaluation of the observation probability is trickier, and depends on the type of feature vectors used. In the case of discrete feature vectors, the log probability estimation is no more than a table lookup. However, in case of continuous feature vectors, the estimation is more complicated, since the state output probability distribution is specified by Gaussian Mixture Densities.(Section II)

- Aggregator nodes: An example is node  $N0$  in figure 7. The dummies(or Aggregators) do not need to perform any kind of multiplication, and do not need to estimate any observation probabilities, since the nodes are non-emitting. The computation for node  $N0$  will be:

$$\log(\psi_0(t+1)) = \max\{\log(\psi_1(t)) + \log(a_{10}), \dots, \log(\psi_6(t)) + \log(a_{60})\}$$

#### B. Pruning

One of the most important techniques for reducing the amount of computation and the recognition time is pruning. A large network will have many nodes and one way to make a significant reduction in the computation needed

is to only propagate tokens having some chance of being the eventual winner. This process is called *pruning*, and is implemented at each time step by keeping a record of the best token overall and de-activating all tokens whose log probabilities fall more than a *beam-width* below the best. If the pruning beam-width is too small, the most likely path might be pruned before its token reaches the end state of the network. This results in a search error. Having a large beam-width on the other hand means keeping a lot of unpromising tokens. Thus, setting the beam-width is a compromise between speed and avoiding search errors. A different type of pruning can be done to bound the total amount of computational resources used. However, the system should be provisioned with enough resources to maintain a reasonably low search error probability due to pruning.

### C. Alternatives to Consider

Considering the presented structure of the recognition network, the most logical way to think about a hardware implementation of the decoder would be to have dedicated processing elements for each of the two types of nodes. Several architectural alternatives can be considered:

- 1) Having a PE for each of the states in the network. This solution implies having different numbers of processing elements for each grammar. An approach looking at implementing recognition networks having one hardware processing element for each node, using FPGA technology, is described in [2]. However, this solution suffers from the drawbacks inherent to FPGAs: high cost, higher power consumption than a custom ASIC. Moreover, the solution would require reconfiguring the FPGA board every time a new grammar and vocabulary is loaded in the recognizer.
- 2) Systolic arrays generally constitute a good match for a wide range of dynamic programming problems. Unfortunately, data aggregation, when the in-degree is unbound (which is the case for our dummy nodes), is difficult and inefficient to implement in such architectures.
- 3) Using a finite set of processing elements performing regular-node computations, and a set of processing elements performing aggregator-type computations. This solution is preferable to the already considered ones, and was chosen as a basis for our design.

### D. Proposed Structure

We propose the following decoder structure (figure 9): The main elements of the decoder are a set of processing elements, each having its own local memory, a set of aggregator units, a shared bus, the general-purpose CPU of the device, and the global memory used by the CPU and the aggregators.

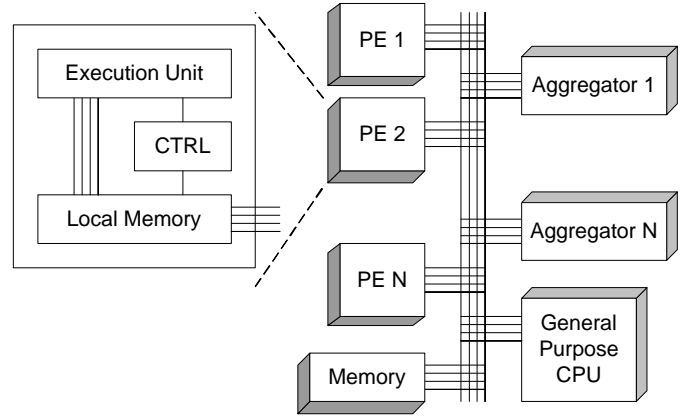


Fig. 9. Architecture of Decoder

**Processing elements:** A processing element consists of an execution unit with local memory for storing the parameters about recognition network nodes along with a control unit. The execution unit comprises of an arithmetic unit, capable of performing fixed-point additions, subtractions and multiplications. These kinds of operations are required to support the regular node computations, as described in section IV-A.

Each PE is assigned a set of nodes and for each node, the PE has to read from memory the values of the tokens it receives from other states, update the probability values with transition and observation likelihoods and choose the winning token storing it back to memory.

One way to do this is to store all the operands for all the nodes stored in a single global memory module but this would incur significant memory contention overheads, since memory accesses account for a large part of the time spent by each PE. We therefore provide each PE with its own local memory, where the operands of the nodes assigned to the PE will be stored. Since the output of one node is needed by other nodes in the next time stage, we cluster nodes into PEs such as to minimize inter-PE communication. The allocation tries to ensure that all token exchanges between regular nodes are within one PE. Only the exchanges from regular nodes to aggregator nodes or the ones between aggregator nodes need interaction between PEs and aggregators. This completely eliminates the communication between PEs since all inter-PE communication now passes through aggregators. The advantage of the approach is that PEs need only access their local memories for all computations. We explain the way scheduling nodes to PEs works in section IV-F.

The contents of the local memory are updated by the general-purpose CPU of the device every time a new language and subsequent recognition network is loaded. The expansion of the recognition network, parameter training - both transition and observation probabilities, and scheduling

of nodes to physical processing elements are all done offline. The contents of each local memory are thus established beforehand, and loaded exactly as received by the networking interface or any other means of communication of the device integrating our hardware recognizer.

**Aggregators:** Each aggregator unit performs the computations for a set of dummy nodes assigned to it.

All aggregator units use the global memory of the device to store their information. The aggregators have the role of aggregating and exchanging data between processing elements. Tokens are pulled from the local memories of the corresponding PEs, compared to find the maximum, and the resulting token is pushed back in the local memories of the PEs that need the result. At each operation, the history, that is kept in the global memory, is updated as well.

For an aggregator unit, the global memory should store the following information for each of the dummy nodes assigned to it:

- **Inputs:** The number of inputs and the pointers to locations of input tokens that can be either in PE local memory (in case of inputs from regular nodes), or in global memory locations (in case of inputs from other aggregators) along with log transition probabilities for each of the inputs.
- **Outputs:** The number of outputs and the pointers to locations where the aggregator must copy the outputs (one for each PE needing the winning token); the result is also stored in the global memory, to be used by all aggregators that need it as input.

Since no aggregator performs any observation probability estimation, the only operations supported by aggregator ALUs are integer and fixed-point additions and subtractions, needed for updating the token value with the transition probability, and computing memory addresses. However, a special-purpose comparator, taking as inputs a number of tokens, and having as a result the winning token, is preferable. Since token comparison is one of the most frequent operations for the aggregators, having a special-purpose token comparator allows this operation to be performed in a single cycle.

As the number of inputs is unbound, varying from case to case, the aggregator must sequentially compare its input tokens, and store the intermediary maximum token in an accumulator. When establishing the number of inputs simultaneously compared, a tradeoff must be made between computation speed and bus-width for the token comparator.

**Bus:** All the elements of the system interact through a single shared bus. The aggregators and CPU are bus masters, reading and writing memory locations. The memory address space is shared between the global memory and the local memories of the PEs. Synchronization among aggregators and PEs to access the local memories and among aggregators to access the global memories can be implemented using bus arbitration.

**CPU:** The general purpose CPU of the device is responsible for loading the initial contents of the local memories and of the global memory portion reserved for speech recognition. The contents correspond to a particular language model and grammar combination and are computed offline. The CPU is also responsible for feeding observations to the local memories of the processing elements, as they are generated by the spectral analyzer.

#### *E. Synchronization of PEs*

For each input observation, each PE has to perform a computation for all the nodes allocated to it. If the network did not have any aggregator nodes, all these operations could be performed in parallel, since each node uses results from the previous iteration of other nodes (including itself). However, the aggregators complicate the problem, since their results for the current iteration are needed in the same iteration by the nodes they deliver tokens to. For example, in figure 7, aggregator  $N0$  must pull the tokens computed by nodes  $N1, N2 \dots N6$  in the previous iteration, add the corresponding transition probability to each of them, then choose the winner and pass it to the nodes that need it in the same iteration.

Our solution is to divide the node computations in each PE into two phases - in the first phase the aggregators as well as the nodes that have no inputs from aggregators perform computations. Once the aggregators have outputs ready, the remaining nodes that needed inputs from aggregators perform their computations. The computation is not completely parallelized, since in the second phase the aggregators are idle, but this drawback has a limited effect, especially due to the fact that nodes executed in phase 2 are a small part of all the nodes, and therefore the duration of phase 2 is much smaller.

#### *F. Scheduling Nodes to PEs*

Since the recognizer has a fixed number of processing elements, and a fixed number of aggregators, we have to assign the logical nodes and aggregators from the recognition network to the physical elements. This scheduling process (done offline by the CPU) must take into account the following constraints:

- If two regular nodes are exchanging tokens, they must be assigned to the same PE.
- The load of the PEs must be balanced, in order to minimize execution time.
- The number of PEs and number of aggregators must be computed such that processing elements need not wait for results from aggregators. After performing a number of experiments, described in detail in section VI, we came to the conclusion that the computation time at the aggregators is orders of magnitude smaller than the computation for each processing element, and therefore a single aggregator unit would be enough to accommodate more than 25 PEs.

- All nodes representing a path between two aggregator nodes must be assigned to the same PE as far as possible. That usually means that states of a single word are assigned to the same PE.

As an example, let's consider the recognition network in figure 10, recognizing any sequence of the words *one* and *two*. The states of word *one* are assigned to PE1, while the states of word *two* and another silence model are assigned to PE2. If, in the initial recognition network, a path between two aggregator nodes is too long to fit in a single processing element, an additional aggregator node is added, splitting the path (see figure 11).

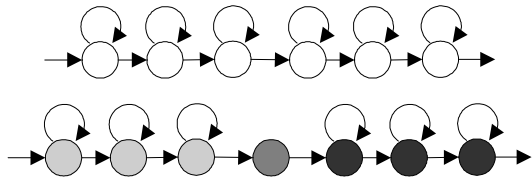


Fig. 11. Splitting a long word

## V. POWER AND PARALLELIZATION

One of our main motivations was to reduce the power at which our speech recognition hardware can run. An effective power reduction mechanism is voltage scaling [10] where the supply voltage of a circuit is reduced in return of higher latency. For CMOS circuits, the chief component of power is dynamic and it can be expressed as  $P_{ref} = kC_{ref}V_{ref}^2f_{ref}$ . The technique used to maintain throughput while reducing the voltage supply is to utilize a parallel architecture. For example if we divide the computation into  $n$  equal processing blocks, we can reduce the clock frequency by a factor of  $n$  but the space occupied by the circuit (capacitance) increases by a factor of 1.15, the 1.15 factor coming from the overhead of extra routing. However, the delay of the circuit is also inversely dependent on the supply voltage obeying roughly:  $T_d \propto V_{dd}/(V_{dd} - V_t)^2$  where  $V_{dd}$  is the supply voltage and  $V_t$  is the threshold voltage. Thus, we can now decrease our supply voltage as we can afford a higher delay. The new power expression is  $P_{par} = kC_{par}V_{par}^2f_{par} = k(1.15nC_{ref})(V_{ref}/k_v)^2(f_{ref}/n)$  where  $k_v$  is the reduction obtained from voltage scaling and it is clear  $P_{par}$  is much smaller than  $P_{ref}$ . From our results, we can see that the parallelization overhead in our design is very low and the extra hardware required for exchanging data among processing elements is small. The only situation where this technique cannot be applied is when we have very heavy constraints on area, but the simplicity of our design allows us to use more silicon.

Unfortunately the gains from voltage reduction become less attractive at very low supply voltages as the delay

increases at a much higher rate, but the operating ranges of current circuits are still above this limit.

## VI. EVALUATION

We evaluate our proposed hardware design, in order to demonstrate the feasibility of speech recognition on power-constrained devices, running at low frequencies. We start by estimating the workload of the system, and the way the workload is balanced among processors. We estimate the number of execution cycles needed for decoding, we compute the total number of operations performed (which is proportional to the power consumption), and we evaluate memory requirements. We continue by looking at the way the system scales by increasing the number of processing elements used, and measuring the communication overhead. We also estimate the benefits of pruning.

### A. Experimental Setup

We simulate the functioning of our decoder using a workload-level event-based simulator. Since we only implement the decoder, we integrate our simulator in the HTK toolchain, providing the rest of the components of a speech recognition system. We use continuous speech coding, providing much higher accuracy than discrete coding. We evaluate our recognition accuracy for phoneme-based recognition.

Our experiments use a speech database of over 6000 speech files, part of which are used for training, part for recognition accuracy estimation. The database has a vocabulary of 37 words, representing all the words required to express any decimal number of arbitrary length. We also implement a scheduling tool, used to allocate nodes from the recognition network to physical processing units. Parameter estimation, recognition network construction and scheduling of nodes to processing elements are performed offline, and the corresponding contents are loaded in the memory of the decoder.

In order to perform the evaluations, we also make the following reasonable assumptions:

- each processing element is equipped with a set of 15 32-bit registers, among which 12 are general and 3 are special purpose. We derived these figures by analyzing the code executed by the processing elements, and minimizing the number of memory accesses as much as possible. The number of registers can be smaller, if traded for some additional memory accesses. The PE has an ALU capable of performing multiplications, additions and subtractions.
- each aggregator unit has a set of 12 general purpose and 10 special 32-bit registers. The ALU is capable of performing additions, subtractions and shifts.
- the clock frequency of the device is in the range 1 - 20 MHz. This frequency range is typical for low-power devices.
- we assume the following cycle counts for arithmetic and memory operations:

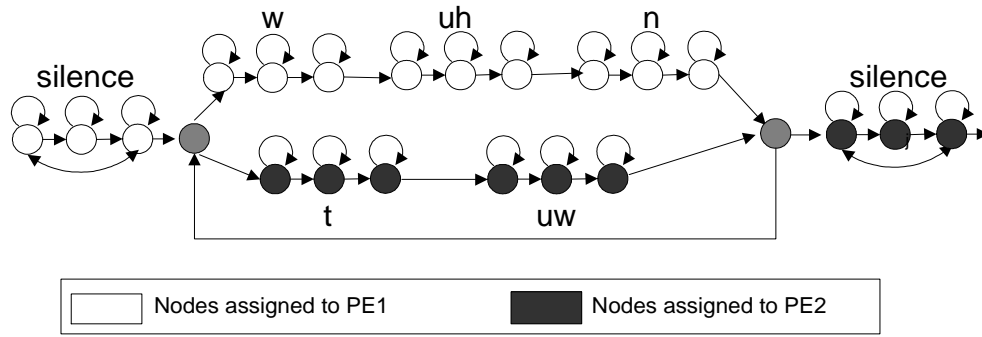


Fig. 10. Example node assignments

Operation:	# cycles
addition	1
subtraction	1
multiplication	10
shift	1
mov	1
jump conditional	1
load/store from local memory	3
remote load/store	15
token comparator	5

The low latency for memory operations is achievable due to the low clock frequency.

- we don't consider any pipelining in executing the operations in the processing elements and aggregators.

### B. Workload Evaluation

The workload of the decoder is dependent on two main factors: the size of the grammar and the size of the feature vector. Both the execution time and the total system workload are proportional to the values of these two parameters.

In order to show the way workload varies with grammar size, we evaluate three grammars: the first containing only 3 possible words, the second containing 18 words and the third 34 words. For each of these grammars, we evaluate the recognition of a 0.51 second speech utterance (having 51 observations), each observation being a feature vector of size 12, running on our decoder with 10 processing elements and one aggregator unit.

Figure 12 presents the number of cycles necessary to perform recognition for each of the grammars. The computation time varies almost linearly with the grammar size. For the largest grammar, 1.04 million cycles are needed, and in order to perform the computation in real-time (in 0.51 seconds, the length of the utterance), we need a clock frequency of approximately 2MHz.

In figure 13, we evaluate the total number of operations performed, in number of cycles, for the same three grammars. As we can see, the total number of operations is an order of magnitude larger than the execution time, due to the 10 PEs executing in parallel.

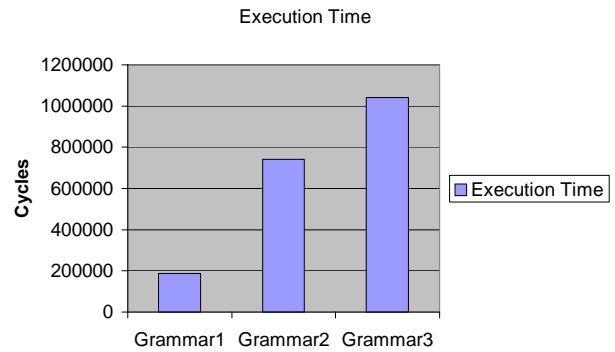


Fig. 12. Execution time for three different grammars

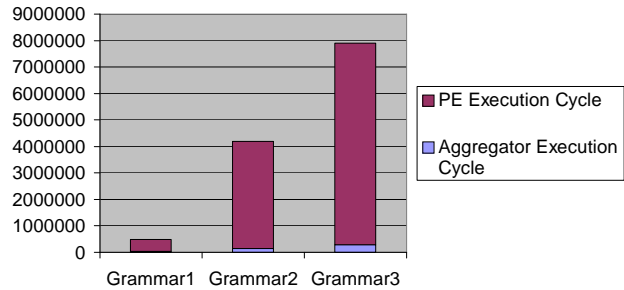


Fig. 13. Total amount of computation for three different grammars

Finally, we evaluate the memory requirements of the decoding process, for the grammars described above (figure 14). The local memories of the processing elements constitute the majority of the total memory requirements. As we can see, the decoder necessitates hundreds of kilobytes of memory, but this number varies with the length of the utterance. However, since the utterances are limited in size, we don't expect more than an order of magnitude difference.

The total amount of computation and execution size are also proportional to the size of the feature vectors (figure

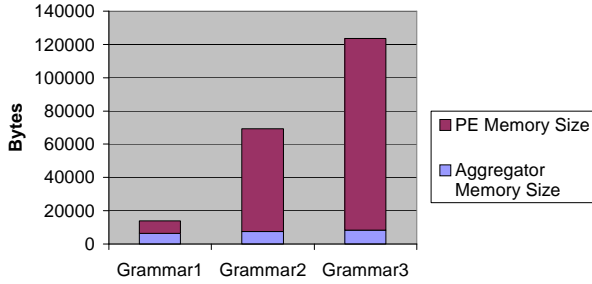


Fig. 14. Memory requirements for three different grammars

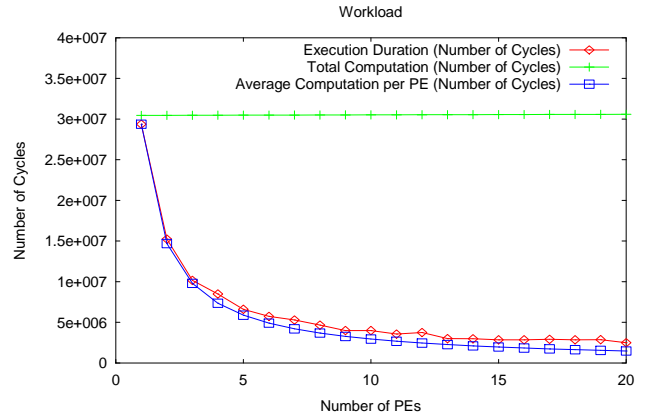


Fig. 16. Scalability of execution time and workload

15).

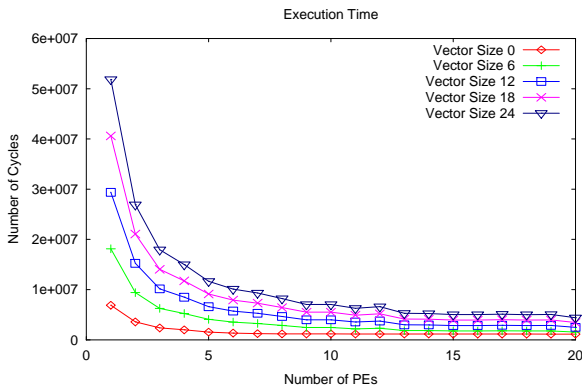


Fig. 15. Execution time for different feature vector sizes, as a function of number of processing elements.

The amount of computation independent of the feature vector size is shown in the plot for a feature vector of size 0. The part of the computation above this line increases linearly with the size of the feature vector (the curves are equally spaced), and represents the amount of computation due to observation probability estimation. As we can see, for large observation sizes, estimating the output probability accounts for most of the computation and execution time spent.

This computation can be reduced if discrete observations are used instead, in which case observation probability estimation reduces to a table lookup, but in this case significant accuracy would be sacrificed.

### C. Scalability

In order to illustrate the scalability properties of our design, let's look at figure 16:

The total amount of computation increases very lightly with the number of processing elements used. This is due to the inherently parallelizable nature of the computation. The small increase is due to the increase in communication overhead, and this remains insignificant even for large number of

processing elements. Since the total amount of computation remains roughly constant, the average computation per processing element indicates the ideal way the execution time should scale. Unfortunately, as the number of PEs increases, our algorithm for scheduling nodes to processing elements does a poor job in balancing the workload, and consequently the execution time is larger than ideal.

Memory size per processing element scales down in a similar manner to the execution time (figure 17). The total memory size increases very slowly with the number of PEs. As we can see, local memory requirements are several times larger than global memory requirements, including history, that is also stored in the global memory.

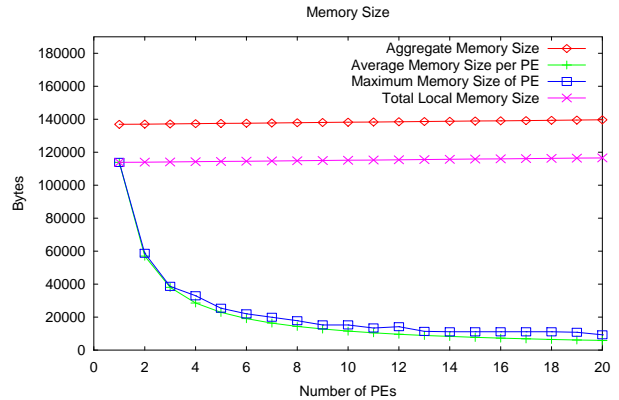


Fig. 17. Scalability of memory requirements.

The speedup efficiency is affected by the poor balancing, but nevertheless it stays around 50% even for 20 processors.

### D. Balancing Number of Aggregators and PEs

After running a few experiments, we quickly realized that aggregators are idle most of the time, and one aggregator can serve alone any reasonable number of processing

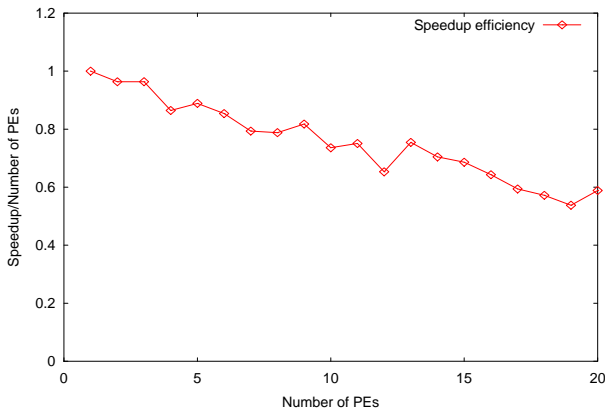


Fig. 18. Speedup efficiency.

elements. We consequently use a single aggregator unit. The following diagram shows that, even for a large number of processors, the aggregator remains idle for more than 50% of the time.

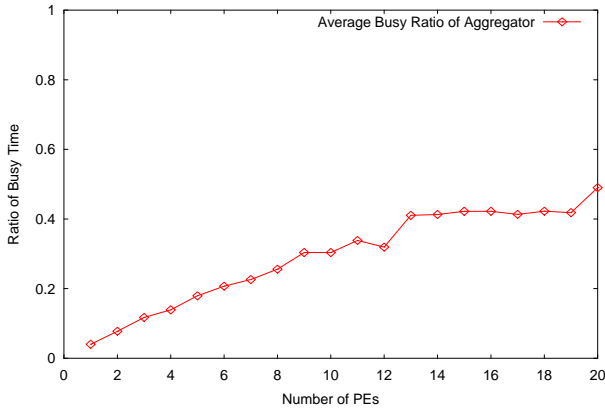


Fig. 19. Ration of busy time to total time for the aggregator.

In fact, the aggregator constitutes a bottleneck for the system only in the initial iterations, when the first token from the start state of the recognition network begins to propagate. In these iterations, the processing elements are mostly idle. However, as soon as more tokens are generated and flow through the network, the aggregator becomes less busy, and the PEs become the bottleneck (see figure 20).

When pruning is applied though, for some iterations it might happen that the computation for the PEs is drastically reduced, while the aggregator's job remains roughly the same, causing the PEs that already finished the execution of phase 1 to wait for aggregator's results before proceeding to phase 2.

### E. Pruning

Pruning is used to reduce the execution time and the total computation, thus reducing the overall power consumption.

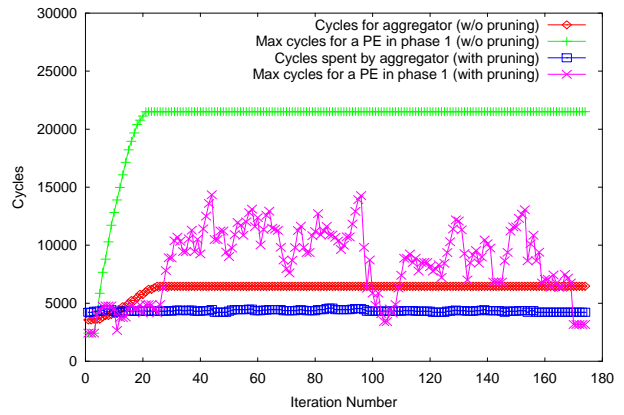


Fig. 20. Number of operations performed in each iteration by aggregators and processing elements in phase 1. This shows when the aggregator becomes bottleneck.

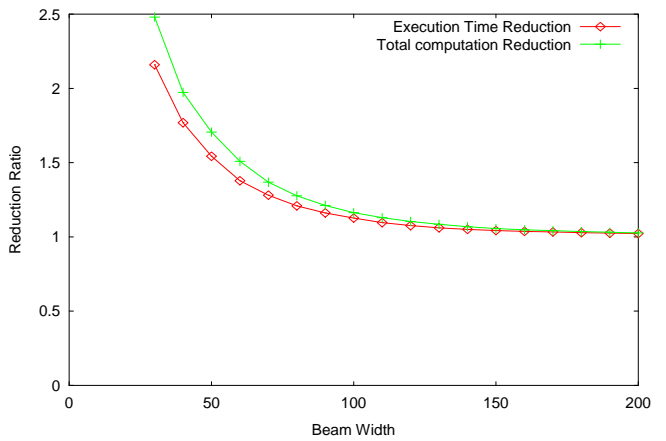


Fig. 21. Execution time and computation reduction using pruning.

As we can see from figure 21, the overall computation can be reduced by a factor of 2.5, if a small beam-width is used. The execution time can also be reduced, but not by the same amount, because tokens with high probabilities tend to be localized, and a single PE having a lot of highly probable tokens is enough to make the given iteration slow, because none of those tokens can be pruned.

### F. Accuracy and Speech coding

Another aim of our study was to determine the optimal speech coding technique that could give us reasonable accuracy while still keeping the computation within bounds. Figure 22 shows best accuracies achievable with different coding schemes with varying feature vector sizes. All the coding schemes are based on the MFCC(Mel Frequency Cepstral Coefficients) with feature sizes from 12 to 39. The key point to observe is that the accuracy does not always depend on the feature vector size - but on the particular data set and training method. We further aim to analyze different

coding mechanisms for accuracy as well as complexity in implementation.

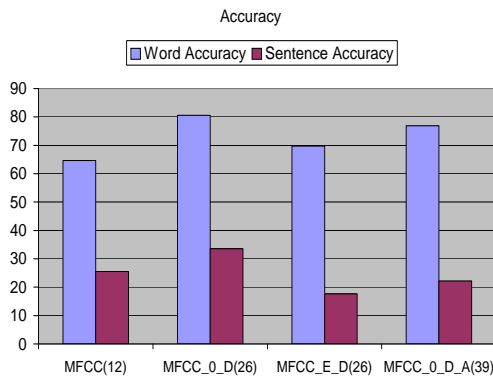


Fig. 22. Sentence and Word Accuracy(%) for different coding techniques

## VII. CONCLUSION

We consider the problem of providing speech recognition functionality in devices facing power, storage and cost constraints. By analyzing different design choices, we conclude that a parallel custom hardware design that performs both signal processing and decoding in hardware is the best solution. Parallelization of computation makes recognition feasible for a large class of power-constrained devices, by enabling real-time decoding at low frequencies and at low voltages, thus decreasing overall power consumption.

The most important contribution of this study is to demonstrate that the decoding process of speech recognition can be easily parallelized using an array of extremely simple processing elements. We show that the space and communication overhead of parallelization is negligible in our proposed design as we exploit data locality by using local memories and decrease the length of the data path by having short buses. We show that our design is scalable and the inherent simplicity of the hardware structure implies small design and development cost, as well as small manufacturing cost. The structure also allows further workload reduction optimizations, such as pruning, that alone reduce the computation by a factor of 2.5. The design is flexible enough to allow the download of HMM models for different languages, different grammars and coding algorithms.

We show that, for grammars of tens of words and observation vectors of tens of values, we can perform real-time recognition at frequencies less than 10 MHz, and requiring less than one megabyte of storage while still maintaining reasonable accuracy.

In the future, to get better estimates of the space/power requirements of our architecture and to verify our claims about the power/cost gains, we plan to implement it in silicon.

## REFERENCES

- [1] S. Young, "The HTK Hidden Markov Model Toolkit: Design and Philosophy," 1993.
- [2] S J Melnikoff, S F Quigley and M J Russell, "Implementing a Simple Continuous Speech Recognition System on an FPGA," 2002.
- [3] Sensory Technologies, "Sensory Speech Products," <http://www.sensoryinc.com/>.
- [4] Conversay Speech Technology Solutions, "Conversay Advanced Symbolic Speech Interface (CASSI)."
- [5] Lim Hong Swee, Texas Instruments, "Implementing Speech Recognition on TMS320C2xx."
- [6] Lorenzo Cali, Francesco Lertora, Monica Besana And Michele Borgatti, "CO-Design Method Enables Speech Recognition SoC," 2001.
- [7] "List of Speech Recognition Hardware Products," <http://www.speech.cs.cmu.edu/comp.speech/Section6/Q6.5.html>, 2002.
- [8] T. S. Anantharaman and R. Bisiani, "A hardware accelerator for speech recognition algorithms," *ACM SIGARCH Computer Architecture News*, vol. 14, no. 2, pp. 216-223, 1986.
- [9] Speech Vision and Robotics Group, Cambridge University Engineering Department, "HTK Tool Kit," <http://htk.eng.cam.ac.uk/>.
- [10] A. Chandrakasan and S. Sheng and R. Brodersen, "Low-Power CMOS Digital Design," 1992.
- [11] L. Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition*. Prentice-Hall, 1993.
- [12] Young. SJ and R. NH and T. JHS, "Token Passing: A Simple Conceptual Model for Connected Speech Recognition Systems," 1989.