A Survey of Hardware Architectures Designed for Speech Recognition

Hsiao-Wuen Hon

August 9, 1991
CMU-CS-91-169[2]

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

In the past few years, there have been many special purpose hardware designs emerging to support speech recognition systems. This paper tries to identify the the system requirements of different spoken language systems' components, such as search and training. Some general design criteria of speech hardware architecture are also presented. Based on these criteria, we survey a variety of notable special purpose computer architectures designed for speech recognition systems and make a paper-and-pencil evaluation of those architecture alternatives.

This research was supported by the Defense Advanced Research Projects Agency (DOD) and monitored by the Space and Naval Warfare Systems Command under Contract N00039-85-C-0163, ARPA Order No. 5167.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

Keyword : speech recognition, spoken language system, Viterbi search, beam search, stack decoding, GSM (Graph Search Machine), ASPEN (AT&T's Systolic Processing Ensemble), MARS (Microprogrammable Accelerator for Rapid Simulation), Search Machine, BEAM, PLUS.

Contents

1. Introduction

Communicating with machines through natural spoken language has always been a dream of human beings. In order to achieve this goal, we need not only an accurate, but also a real-time speech recognition system. Speech recognition has been an active research area for quite some time, and a good deal of progress has been made in the last couple of decades. In the last three to five years, the Hidden Markov Model (HMM) [26, 13, 23, 34, 43, 48] has proven to be a popular and effective technique for the implementation of speech recognition systems. However, the ultimate goal of highly accurate large-vocabulary continuous speech recognition with real-time response is hard to achieve without special hardware support because of the large computational and memory requirements of the HMM approach.

In order to speed up the speech recognition process, many special purpose hardware designs have emerged for speech recognition systems in the last couple of years. They range from fully custom designed machines, to systolic machines, to pipeline machines, to shared memory multiprocessor machines, to message passing machines, to distributed multiprocessor machines. Of course, every architecture has its own pros and cons. Some architectures are much faster, while some are easy to program, easy to expand, cheap to build, or flexible for different kinds of applications, etc. This paper attempts to describe those features and provide a survey of some notable systems developed recently. We would like this document to serve as a future reference for people who may contemplate building new speech architectures or for those who would like to choose among the architecture alternatives that are available.

The paper is organized as follows. Section 2 characterizes the general computation and memory requirements for a variety of speech related algorithms , like recognition and training. Section 3 addresses the general design criteria of hardware architectures for speech recognition systems. The paper-and-pencil evaluation and comparison of a variety of notable architectural alternatives are presented in Section 4.

2. General System Requirements for Speech Recognition Systems

Although this paper focuses on HMM based systems, it doesn't necessarily exclude other approaches, like knowledge-based systems [52] or neural network based systems [51]. Nevertheless, since the HMM has been a predominant approach for continuous speech recognition, almost all

of the hardware architecture designs for speech recognition are aimed at HMM based systems. Actually, some architectures mentioned in this paper could be extended to non-HMM systems with small modifications.

By looking into the spectrum of the whole speech recognition system, front-end signal processing [33, 29] for the transformation of the input signal into a form suitable for analysis is of no concern from the computational point of view. Although *auditory models* [14, 32, 31, 1] suggested by some speech researchers require large computation, their superiority in continuous speech recognition has not been shown. Commercial digit signal processors can handily generate the two most widely used speech signal representations: FFT and LPC [33] in real time. On the other hand, the search part requires large computation and therefore becomes the most challenging part of the architecture design in order to achieve real-time performance to produce a practical speech recognition system. Moreover, although the training process whose responsibility is to build accurate models for recognition doesn't require real-time performance in general, it is still important for the system to speed up the training process so as to reduce the time required for model development.

In the following, we will describe and characterize the general computation memory requirements for a variety of speech related algorithms, including Viterbi search, stack decoding and training. Since parallel(including pipeline) architecture is used to speed up intense computation most frequently, we will pay more attention on parallel implementation of such algorithms.

2.1. Search

2.1.1. Viterbi Search

The Viterbi Search [50] is the most popular search algorithm for continuous speech recognition because of its efficiency. It has been successfully applied in many speech recognition systems [27, 35, 13, 48, 34, 42]. The Viterbi Search is a time synchronous search algorithm that completely processes all input up to time t before going on to time $t + 1$. For time t, each state is updated by the best score from states at time $t - 1$. From this, the most probable state sequence can be found by backtracking at the end of the search.

A full Viterbi search is quite efficient for moderate tasks [7]. However, for large tasks, it can be very time consuming. A straightforward way to prune the search space is the beam search

[30, 49, 36]. Instead of retaining all candidates at every time frame, a threshold T is used to maintain only a group of likely candidates. The use of the beam search alleviates the necessity of enumerating all of the states and can lead to substantial savings in computation with little or no loss of accuracy. For example, in the implementation of SPHINX [25], they found it is possible to prune 80% - 90% of the hypotheses, without any loss in accuracy.

There are several advantages to using the Viterbi search algorithm. First, if we use a logarithmic representation of the HMM probabilities, Viterbi search is extremely efficient because multiplications of probabilities become additions. Second, because of its time-synchronous nature, it is very easy to modify the time-synchronous Viterbi algorithm into a beam search and it will also be suitable for parallel implementation if necessary.

Although the actual implementations of the Viterbi search algorithm will vary, it can roughly be broken down into the following two modules:

- The **word expansion module** performs Viterbi search and finds the probabilities for words in the vocabulary matching the speech signal.
- The **grammar expansion module** controls the word expansion module by telling it what words to process in the next time frame based on grammar constraints.

The DARPA resource management task (see [25] for a description of this task), which is a 1,000−word, perplexity 60 task, has been used as a common benchmark task by the speech community since 1987. As a result, many architecture designs aimed to achieve this task in real-time. We will use the resource management task as our benchmark test to evaluate different architecture designs in section 4. In order to do that, we must first analyze the time and space requirement for Viterbi search in the resource management task. Let's take SPHINX's [27] implementation for an example, without loss of generality. In the SPHINX system, allophones are described with seven-state models and words are sequences of (on average) six allophones. Therefore, each word requires about 40 states. Resource management used a statistical grammar which specifies the legal word pairs. In a regular Viterbi algorithm, the word expansion module needs to update all the states at every frame (10 ms), so for the resource management task, this would require updating about 40,000 states every 10ms, or about 80,000 transitions, since there are, on average, two predecessors for each state. There are about 7 instructions required for each transition in the SPHINX system, which used a discrete HMM approach. (For continuous HMM

approaches, the number of instructions required is at least one order of magnitude greater than that for the discrete HMM approach. Please see section 2.1.3 for details) This is quite a lot of computation. Fortunately, because of its time-synchronous nature, the word expansion module can be implemented in parallel efficiently. The other way to reduce the computational load is to use the beam search described earlier. For example, a beam search could reduce the number of states that need to be updated at every frame to about 4,000 (8,000 transitions) instead of 40,000 (80,000 transitions) in the SPHINX's implementation. On the other hand, the computational requirements of the grammar expansion module depend on the complexity of grammar. For example, the resource management task with a word-pair grammar of perplexity 60 will need to process 1,000 * 60 grammar transitions for non-pruned Viterbi search. Of course, the beam search can again be used here to reduce the computational requirements. If the Viterbi search algorithm (with or without beam pruning) is implemented on a single-processor general-purpose machine, the word expansion module generally consumes about 70% to 80% of the total search time(and up to 95% if using continuous densities), while the grammar expansion module consumes about 20% to 30%. Therefore, most system designs are aimed at speeding up the word expansion module.

While the speed of commercial general purpose processors continues to improve every year (at a rate of two to three times per year), the speed of memory access does not increase at the same rate. [20, 39] Unfortunately, the word expansion and grammar expansion modules require access to a large amount of memory with a high bandwidth, and therefore the Viterbi search is mainly a memory-bound task. Because of the frame-based nature of the Viterbi search, it exhibits a very poor cache hit rate. Therefore, memory bandwidth is probably the major bottleneck in Viterbi search. For example, The DARPA resource management task requires an average memory bandwidth of 40 Mbytes per second for SPHINX [9] if it is executed in real time. This figure could even be larger because most systems tend to use more sophisticated models in order to achieve better performance. Most commercial bus-based shared memory multi-processor machines are not a good solution for Viterbi search because they work well only with algorithms that have a high cache hit-rate and the serious memory contention will slow down the processors when a few processors (more than 5) running in parallel.

The other potential problem for parallel implementation of Viterbi search is the substantial amount of synchronization because Viterbi algorithm reveals a very small granularity. For example, the DARPA resource management task needs about 2 μs between synchronizations even with the beam search. Although beam search could substantially reduced computation, it also introduces the problem of load-balancing because it retains an unknown pattern of the most promising states

and therefore requires more complicated synchronization among processors.

2.1.2. Stack Decoding

Stack decoding [3] is a best-first search algorithm for continuous speech recognition. It is derived from the A^* algorithm [37] used in artificial intelligence. Stack decoding is not time-synchronous, but extends paths of different lengths. Stack decoding constructs a search tree from a state graph S that is specified by the language model. The states in the graph represent the states in the language, and the branches in the search tree are words that cause transition from one language state to another. The search begins by adding the initial states of the language with empty hypothesis to the *OPEN* list. Then every time the best hypothesis is popped out from the *OPEN* list, all paths from it are extended, evaluated, and placed back in the *OPEN* list. This search continues until a complete path that is guaranteed to be no worse than all paths in the *OPEN* list has been found. In selecting the *best* hypothesis, we need an evaluation function that estimates the score of the complete path as the sum of the actual score of the partial path and the expected score of the remaining path. If the expected score of the remaining path is always an underestimate of the actual score, then the solution found will be optimal. This feature is called the admissibility (*optimality*) feature of the A^* search. [38]

Viterbi search is a graph search, and paths cannot be summed because they may have different word histories. Stack decoding is a tree search, so each node in OPEN has a unique history, and paths can be summed within word hypotheses when a word is extended. Therefore, with stack decoding, it is possible to find the optimal word sequence with multiple beginning and ending times which is more consistent with the training algorithm [8], than the optimal state sequence. Moreover, Viterbi search computes over the entire search space at all times (although beam search can reduce the search space), while stack decoding only computes with the best hypothesis in each expansion. Since the memory requirement of dynamic expansion is much less than that of Viterbi search, stack decoding is likely to be able to deal with very large vocabularies and more complicated grammars.

While the aforementioned properties of stack decoding are very attractive, many implementation problems arise for stack decoding. First, it is very hard to compare the probabilities of partial paths with different lengths. Direct comparison of probabilities will cause the search to always favor shorter paths, resulting in a full breadth-first search. On the other hand, since the acoustic

probabilities being compared in Viterbi search are always based on the same partial path, which is not the case in stack decoding, stack decoding needs to perform some normalization on the probabilities. Unfortunately, there is no easy way to find such a normalization. Second, the evaluation function that guarantees the admissibility of the algorithm is difficult to find for speech recognition because of uncertainty about the remaining part of speech. A function that is guaranteed to underestimate will result in a very large *OPEN* list, so a heuristic function that may overestimate has to be used to evaluate the remaining paths. Of course, this invalidates the admissibility of the algorithm. Even with an overestimating evaluation function, the *OPEN* list will still be too large. Therefore, two other types of pruning are used to reduce the search burden:

1. A fast-match phrase that extends only a small fraction of most promising paths from a partial path in order to get a list of most likely following words waiting for detailed matching. [4]
2. The use of a stack (similar to the beam in Viterbi search) that saves only a fixed number of hypotheses in the *OPEN* list.

Finally, consideration of multiple beginning and ending time of words is very tricky because it is too costly to try all possible beginning and ending times. Some heuristic must be used to determine the boundary regions.

Based on the problems described above, stack decoding is considerably more difficult to implement. Thus, very limited effort has been invested on the stack decoding for continuous speech recognition systems [6, 41]. There are even fewer hardware architectures aimed at stack decoding. Although all speech architectures mentioned in this paper are aimed at Viterbi search, we would also like to analyze the possibility of exploring stack decoding. In order to do this, let's now try to characterize the potential system requirements of stack decoding.

Unlike Viterbi search, a parallel implementation of stack decoding is not so straightforward since it is not time-synchronous. One obvious parallel implementation could be performed by popping N-best hypotheses from the stack and sending each to different processors. Once each processor completes one level of expansion(from one language state to another language state), the new hypotheses will be inserted back into the stack or merged with existing states if necessary. Since stack decoding is a best-first search, some early actions taken by executing non-best hypotheses in parallel might be proven to be unnecessary later and therefore it might be a waste of time to explore the less promising hypotheses. Based on simulation results [40], the best complete path usually comes from the 10-best hypotheses. From such a point of view, this kind of decomposition seems

not to be suitable for massive parallel implementations. The second type of parallel decomposition could be done when performing the search on the best hypotheses. In general, there are many language transitions for a language state depending on the perplexity of the grammar. For example, there are on the average 60 grammar transitions per language state for a grammar of perplexity 60. It is ideal to have a master processor popping out the best hypothesis and handing out the different word evaluations to different slave processors according to the grammar transitions. After the slave processors finish word evaluation, the new hypotheses along with the score information are sent back to the master processor and the master processor will finish the bookkeeping, merging and finally inserting the new hypotheses back into the stack. This decomposition allows not only for massive parallelism, but also for medium granularity because there are time-synchronous constraints. On the other hand, hypotheses are extended one word at a time instead of being extended one frame at a time in stack decoding, and different words tend to have different lengths, so processors are not likely to terminate at the same time when running word evaluation in parallel. Therefore dynamic load balancing is very crucial to implement stack decoding in parallel efficiently. Moreover, the communication between processors is more complicated than Viterbi search because it happens almost everywhere and is no longer time-synchronous. Fortunately, the memory bandwidth issue seems to be a less serious problem for stack decoding. Since stack decoders do word-based evaluation instead of frame-based evaluation, in stack decoding, it is not necessary to deal with the whole search space that Viterbi is always likely to access. Processors would have a relatively good temporal and spatial locality and therefore stack decoding would achieve a much higher cache hit-rate than Viterbi search.

The other related issue in stack decoding is the use of fast match. Since stack decoding doesn't have the ability to prune unpromising word hypotheses until the completion of the whole word evaluation, it becomes increasingly important to find a short list of promising candidate words quickly and cheaply (fast match) before performing slow and expensive detailed matches, as the vocabulary increases in size. Because it must be quick and cheap, a parallel implementation is ideal for fast match. Like implementing the word evaluation in parallel, this parallelism tends to have medium granularity and high cache hit-rate. In addition, we could have detailed acoustic matching for the most promising hypothesis and use fast match to find the short list of words for the second most promising hypothesis to do detailed matching next running in parallel, and so on. However, the cost of complex synchronization is the inevitable price to pay for doing this kind of hybrid parallelism.

2.1.3. Continuous vs. Discrete Density

The other important feature which would affect the design of speech architecture is the type of density functions embedded in the HMM representation. One is the *discrete* density function [46] which models the output probability for each output symbol explicitly. In association with discrete density HMMs, each frame of speech must be represented by a symbol from a finite alphabet through vector quantization (VQ) [29]. VQ is a data compression technique that first identifies a set of prototype vectors from training data. Then speech input is converted from a multi-dimensional real feature vector of, say FFT or LPC coefficients, to a symbol that represents the best-matching prototype vector. The other is the *continuous* density function. By pre-assuming certain properties of the output distributions, the output probability of observed speech can be attained according to the distribution formulas. The most frequently used continuous density is the multivariate Gaussian density [46]. With the multivariate Gaussian density, the output probability density function (pdf) is described by a mean vector and a covariance matrix. To reduce computation, the covariance matrix is sometimes assumed to be diagonal (all the off-diagonal terms are zeros). Moreover, mixture densities [47] are often used to improve the modeling because a single-mixture diagonal Gaussian density function does not adequately represent certain speech parameters [47].

The use of continuous HMMs requires heavy computation. By using discrete HMMs, computing the output probability of an observed speech frame during the search is merely vector quantization and table-lookup. On the other hand, by using the continuous HMMs, more memory accesses and many more instructions are required for every transition even with the simplest single-mixture diagonal Gaussian density function. For example, every single-mixture diagonal Gaussian density function in the AT&T system requires more than 72 multiplications and 36 additions. The use of multiple-mixture or full covariance would definitely further increase the computational requirement of speech recognition with continuous HMMs.

In order to compare the computation requirement of discrete and continuous densities, two pseudo codes for computing observation probabilities from discrete and continuous densities are given as follows. The pseudo code for discrete densities is based on the implementation in SPHINX and that for continuous densities is based on the semi-continuous SPHINX system [22] which resembles a 4-mixture multivariate diagonal Gaussian density function. Logarithmic probabilities are used in both cases. Suppose there are 3 different codebooks and the size for each one is 256. As we can see in the first part (quantization), continuous densities need about twice as many multiplications and additions as the discrete ones. However, in the second part

(output probability generation), while discrete densities only need 3 additions for each distribution, continuous densities need 3 * 4 * 2 more additions, 3 * 4 more exponentiations and 3 more logarithms which are very expensive for each distribution. Be aware that the computation for discrete densities is fixed when the number of the codebooks and the sizes of codebooks remain unchanged. On the other hand, the computation for continuous densities could be increased by using more mixtures or more Gaussian densities even when the number of codebooks and the sizes of the codebooks remain unchanged. For example, the AT&T system [24], which uses more mixtures (more than 60) and more Gaussian densities (more than 9,000), could potentially require much more computation.

Discrete Density:

```
for each frame {
    for each codebook K_i {
        min = 0;        B_i = 0;
        for each code C_j in K_i {
            dist = 0;
            for each dimension k in C_j {
                dist = dist + (cep[k] - C_j[k])^2;
                /* cep[] is the cepstra vector and C_j[] is the centroid vector for C_j */
            }
            if (dist < min) {
              B_i = j;        min = dist;
            }
        }
    }

    for each distribution i {
        Prob_i = 0;
        for each codebook j {
            Prob_i = Prob_i + Out_i,j[B_j];
        }
    }
}
```

Continuous Density:

```
for each frame {
    for each codebook K_i {
        for each code C_j in K_i {
            Gauss_{i,j}.prob = 0;        Gauss_{i,j}.code = j;
            for each dimension k in C_j {
                Gauss_{i,j}.prob = Gauss_{i,j}.prob − (cep[k] − C_j[k])^2 * Var_j[k];
            /* cep[] is the cepstra vector, C_j[] is the mean vector and Var_j[] is the covariance vector of C_j */
            }
            Gauss_{i,j}.prob = Gauss_{i,j}.prob + Det_{i,j};
            /* Det_j is the determinant of Gaussian pdf Gaussi, j */
        }
    }

    for each codebook K_i
        SORT Gauss[i, 1..size_of_codebook_K_i] according to field prob

    for each distribution i {
        Prob_i = 0;
        for each codebook j {
            temp = 1;
            for each mixture k {
                temp = temp + exp(Out_{i,j}[Gauss_{j,k}.code] + Gauss_{j,k}.prob);
                /* Out_{i,j}[] is the mixture weight vector for distribution i, codebook j */
            }
            Prob_i = Prob_i + log(temp);
        }
    }
}
```

In spite of the heavy computational requirements of continuous HMMs, it doesn't necessarily present as tough a problem as we might have thought at first glance. First of all, thanks to advanced VLSI technology, multiplication can be executed as fast as addition; the exponentiation

and logarithm could also be executed effectively by mathematic co-processors. Second, the computation of observation probabilities could be logically separated from the search part, so they can be executed in parallel and the search part only needs to access the observation probability result buffer filled by the observation probability generator. This functional separation could also be used to speed up large vocabulary Viterbi search on a single processor machine. For example, while there are about 2,000 distributions in the SPHINX system, it needs to update about 8,000 transitions at every frame (see Section 2.1.1). Therefore, without functional separation, it will need to compute 8,000 distributions instead of 2,000 and many of them are actually re-computed several times. The recent SPHINX system [22] used a semi-continuous density function which resembles a 4-mixture multivariate Gaussian density function and resulted in only a 3-5 times slower recognition performance than the discrete SPHINX, while the performance is about 3 times slower without the functional separation. Moreover, the observation probability generator is a sequence of vector-style operations which could be implemented in parallel and involve almost no synchronization and load-balancing. The vector-style observation probability generator could even be accelerated by vector processing machines like the CRAY and Alliance.

2.2. Training

In comparison with recognition, training is a much less serious problem because the paths to search in training are always known. A good implementation on a commercial single processor workstations (SUNs, DEC Stations, NEXT) could easily perform the job in real-time. However, the size of the training database normally consists thousands of utterances constituting hours of speech. For example, the DARPA speaker-independent resource management training database[44] consists of about 4,000 utterances and CMU's general English training database consists of more than 20,000 utterances [21]. The training could also consists of several iterations and several phases, such as mono-phone training and triphone training. Besides, the discriminative training schemes (such as Maximum Mutual Information training, corrective training) [11, 5] could add a lot of phases to the training process. Therefore, it would be nice for the speech hardware architecture to accelerate the training process as well.

As mentioned before, training on one utterance is basically a one-path parametric re-estimation and thus is a sequential behavior which could not benefit from a parallel implementation. However, since the training database contains many utterances, it is possible to train different utterances in parallel. This decomposition is very easy and requires only a small amount of synchronization (the

granularity is relatively large, only 1-2 seconds between synchronization).

Although the memory bandwidth requirement for training is much less than that for recognition, the total memory needed for training could be much more because training normally needs several copies of the HMM's. Since the training on different utterances would require accessing different models, the visibility of all the models for training processor would be essential for speed-up. If the primary memory can't hold all the spaces needed for training, disk swapping would be fatal to the training process. Fortunately, swapping between disks and primary memory, and between primary memory and cache only happens when completing the training on the current utterance and beginning to train on the next utterances. We could arrange the training database so that the differences between consecutive utterances are minimized, therefore requiring less page-in's and page-out's. The other possibility is to create one process to pre-fetch the required data for the next utterances when training on the current utterance [2]. Even if the pre-fetch requires page-in or page-out, the fetch operation would not affect the training on the current utterance since the CPU operation and IO operation are interleaved.

3. Design Criteria of Speech Hardware Architectures

In this section, we would like to give some of the desired criteria for speech hardware architecture designs. A good architecture design doesn't necessarily need to cover all the criteria we state because different architecture designs have their own goals. However, we would like to list some criteria which many systems try to achieve based on the requirements of a speech recognition system.

- **Speed** - First of all, speed is the main motivation for designing and building a special purpose hardware architecture for a speech recognition system when general purpose machines can not perform the task effectively. Since the ultimate goal of automatic speech recognition is to communicate with machines through natural spoken language, real-time performance is essential for success.

- **Scalability** - Although most system designs have some specific task in mind (such as a resource management task), it would be desirable for the system to be extensible. Because of the continuous development of speech technology, we are able to handle more and more difficult tasks. An extensible system would be a more useful tool for speech researchers than

a stand-alone system for some limited applications. The first way to enhance the performance of a system is to use a faster processor or clock, but this is a very limited approach and not trivial to implement. The other is to use more processors in parallel systems. However, this is not so straightforward because speech search is a memory bound algorithm. Using more processors might saturate the bandwidth of system very soon and therefore might degrade the system instead. For example, the SPHINX system cannot take advantage of more than five Encore Multimax processors (Encore is a commercial bus-based shared-memory machine) because the memory bus bandwidth is exhausted with only a few (5 or 6) processors.

- **Flexibility** - There are many different configurations of speech models (continuous vs. discrete density) and different search algorithms (Viterbi vs. stack decoding). Although a few mixed comparisons have been done, the superiority of one over the other is still an open question. Therefore, a good hardware design might need to accommodate different configurations and different algorithms. In order to make the special purpose hardware helpful for speech researchers as development machines, it must also be feasible for all the related speech algorithms, such as training and natural language processing. Unfortunately, those speech algorithms are quite different, though they share some fundamental operations. Flexibility will remain a big challenge for speech architecture design because it is well known that flexibility and speed are two tradeoff factors in hardware design.

- **Programming Environment** - The other important issue to make the special-purpose hardware more useful for speech researchers is the programming environment. General purpose languages are favored over microprogramming languages. Systems that require separate programming languages for different parts, like a signal processing subsystem, a word expansion module, a grammar expansion module, etc, would prevent people from using them regularly. Since most system designs are based on parallelism, the other important issue in programming environments are the parallel programming and debugging tools. Writing correct parallel programs is never easy. Therefore, the tool that enables the users to execute and debug their parallel programs on the host machine or single processor machine will be extremely helpful in reducing the development time.

- **Development Cost** - While the speed of computers continues to increase, the price of computers is decreasing. A simple speech task (with a small vocabulary or highly constrained grammar) could be handled in real time by a commercial general purpose machine. Therefore, custom machines are adequate only if their cost is not so astonishing in comparison with the cost of general purpose machines. On the other hand, general purpose machines are continuing to improve at a very high rate (2-3 times per year) and eventually will perform the

same task sufficiently (the fate of all architectures). Moreover, speech modeling techniques keep changing, so the development of special hardware architectures is meaningful only when it is developed fast enough to catch up with the current hardware technology and speech modeling techniques.

- **Miniaturization** - Since the development of VLSI techniques, a new criterion has arisen for custom architecture design, namely *miniaturization*. For some applications, requirements are stated not in MegaFLOPS, but in MegaFLOPS per cubic foot. Therefore, a good architecture design should also take into account the suitability of miniaturization via VLSI and packaging.

4. Survey of Some Notable Systems

4.1. AT&T's Graph Search Machine

AT&T's Graph Search Machine (GSM) [17] is a custom designed VLSI chip that is aimed at speeding up the kernel functions used in dynamic programming (or Viterbi search). The architecture is designed around a fast accumulate-minimum selector function that complements the classical multiply-accumulate function in many signal processing devices. The function is optimized by special design of the datapath to allow a pipeline configuration. Also like many DSP architectures, the GSM uses both on-chip and off-chip memories to increase performance. While on-chip memory ($32 * 32$ bits) is used as a program "cache", off-chip memory ($64K * 16$ bits) is used to hold data as well as programs. Although the 68-pin GSM chip runs at a clock speed of 8 MHz, the GSM can search 500 templates in real time when using dynamic programming on word template matching. It is several times faster than typical microprocessors.

To use the GSM for continuous speech recognition by the HMM approach, the limitation is on the local memory, but not on the processor. For example, the SPHINX system has roughly 600 HMM's and each HMM occupies approximately 10 Kbytes [25] for the DRAPA resource management task, so the local memory of each GSM can hold at most 12 HMM's. Therefore, one must have at least 50 GSM's running in parallel in order to achieve the resource management task in real time. Since the single-bus connection among the GSM's and no shared memory support prevent it from executing complicated communication and dynamic load balancing, a beam search can not help in this case.

The GSM could indeed be a component of every architecture design because one could take advantage of fast execution of the kernel function of Viterbi search. As indicated earlier, the bottlenecks for the GSM are the memory and communication requirements. When using the GSM in a parallel mode for larger tasks, the real performance will mostly depend on the architecture built on top of it because there is no support for any synchronization or shared memory management. Moreover, its superiority is limited to Viterbi-like algorithms. For example, the kernel function of stack decoding and training, which is a summation of all the possible paths instead of the selection of the minimum path, would not be able to take advantage of the GSM. Considering natural language processing, the GSM is only good for finite state type grammars, but not for most other parser type grammars.

On the other hand, to take full advantage of the GSM, one must program it in the GSM microcode in order to execute up to 11 instructions in parallel. It would actually increase the burden for users unless there is such a good compiler for the GSM that will generate the nice parallel stage code for application programs. Therefore, from the user's point of view, custom designed search processors are attractive only when the speed of them is much faster than general-purpose processors.

4.2. SRI-Berkeley's Speech Search Machine

SRI-Berkeley's speech search machine [45] represents a fully custom-designed machine for speech recognition. The system is functionally decomposed into three separate modules : a front-end module, a word-processor module and a grammar module. The system is partitioned in this way to meet the different processing needs of the different modules. The front-end module is of no concern because of its simplicity. The word-processor is implemented with a more complex single-processor pipelined architecture that has several memory inputs operating concurrently to avoid the memory contention problem of parallel design. With the help of prefetching required data into on-chip memory and three concurrent data paths, it can execute a one-state Viterbi evaluation in a single clock cycle. On the other hand, the grammar module that requires many simple computations is implemented with a set of concurrent processors. With the help of partitioning the whole data memory space into different sections supported by different processors, the grammar module is able to cope with the memory bottleneck and therefore, it can process one grammatical transition every clock cycle.

The obvious advantage of this system is the speed. With a 10 Mhz clock and a four-processor

grammar module, SRI-Berkeley's search machine can perform a full Viterbi search without pruning in real time on a 5,000-word, perplexity 80 task which is larger than the Resource Management task (1,000 words, perplexity 60). By projection, it can execute the Resource Management task in real time with only a 5 Mhz clock and a two-processor grammar module.

The SRI-Berkeley search machine suffers from the inflexibility of a custom-designed architecture. For example, it would need major architecture changes in both the word-processor and the grammar modules to incorporate beam pruning. In the word-processor module, its pipelined feature heavily relies on the uniform data paths of predecessor probabilities and HMM parameters. Beam pruning would definitely interrupt those data paths making their performance unpredictable. Finally, beam pruning also presents the problem of dynamic load balancing among the grammar processors which were previously balanced in the non-pruned Viterbi search through static data partition. Other inflexibilities of this system include :

- It is hard to change the topology and configuration of HMM models. For example, the topology with more than three predecessors for a state will require more than one cycle to update the state. Moreover, The system handles only up to four discrete output densities. The use of more discrete output densities or continuous densities will force re-design of the word-processor module.
- It can only accept a statistical grammar, such as bigram, trigram, etc. The use of other types of grammars would require re-design of the grammar module.
- It is almost impossible to execute other types of speech algorithms, such as stack decoding, natural language processing, etc, because all the design decisions are tailored to Viterbi search. For example, stack decoding, and parsing type natural language processing are definitely not suitable for pipeline design because they don't have a time-synchronous nature. However, SRI-Bekeley's search machine would be suitable for training with only a few changes because the basic difference between the forward-backward training and Viterbi search is the summation vs. selection.

The SRI-Berkeley's search machine seems not to be scalable because the word-processor module is not a parallel design although the grammar module could be scalable. The only possible scalable scheme is to use fast chips or clocks. However, it is very limited and might need some re-design because of the a relatively slow memory access rate. However, scalability might not present a serious problem for it because it can handle relatively larger task than most other system already.

Like most custom-design systems, the SRI-Berkeley system would need a considerably longer development time and therefore could be more expensive than other architectures. Finally, since the designs of the 3 modules are different, it could make the programming environment more complicated. One might expect that different programming languages are necessary for different modules and result in long system developing and debugging stages.

In conclusion, SRI-Berkeley's system demonstrates that a fully custom-designed machine can perform a Viterbi search one or two orders of magnitude better than the state-of-the-art general purpose machines. Although it might lose flexibility and be expensive, its ability to handle relatively large and difficult tasks would hopefully overshadow those disadvantages.

4.3. AT&T's ASPEN Tree-Machine

AT&T's ASPEN (AT&T's Systolic Processing Ensemble) [19, 18] is a medium grain tree-structured parallel architecture, scalable to thousands of processing elements (PE's). Each PE has one parent and two children, and consists of a semi-general purpose processor (8-MFLOP AT&T DSP32) and 64 Kbytes of local memory. A board-level module comprises 8 PE's on an 8"x13" board. These boards are interconnected to form a larger binary tree according to Leiserson's expansion scheme. [28] To apply Viterbi search to ASPEN, the HMM's are distributed among the PE's first. For each time frame, the host receives the speech vector and broadcasts them and the initial Viterbi scores from the previous frames to all the PE's, and a within-model Viterbi search is then executed within the PE's. Finally, the between-model Viterbi search and the initial scores for the next frame is done via resolve/report/broadcast functions which are cooperated by the host and PE's.

For the benchmark test, AT&T reported that a 127-PE (16 boards), 1,000-MFLOP configuration was needed for DARPA resource management task, if the 7-state and 3-Gaussian-mixture whole-word models were used. Although this figure is required by the continuous HMM approach, the less computation-loaded discrete HMM approach would not reduce the requirement. Since the local memory for each PE is only 64 Kbytes and a discrete HMM occupies about 10K (in the SPHINX system), the discrete HMM approach with 600 models would indeed require about 100 PE's in order to distribute all the models across PE's unless one uses less complicated models.

A tree-machine is an attractive architecture for Viterbi search because of its great scalability characteristics. The PE's remain the same as the tree grows. The interconnection overhead is small because the communicate on complexity grows only as $\log N$, where N is the number of

PE's in the tree-machine. Therefore, as the task becomes larger, the number of processors can be increased by adding more PE's to the binary tree. Moreover, the inter-processor communications within the tree controlled by the host are independent of the size of the tree and each PE basically runs the same code. Therefore, the source codes for an application are essentially unchanged when additional PE's are added. This software scalability make a tree machine easy to program. Because of both hardware and software scalability, one can take big advantage of the computation power of many PE's. For example, AT&T used the computation power of ASPEN to deal with continuous densities with many Gaussian mixtures.

While the tree-machine has a relatively small communication time, it has a fixed and limited communication bandwidth among PE's which prevents it from performing complicated inter-processor communication. Moreover, the tree interconnection can only be effective for a tree communication pattern because there is no routing hardware. Unfortunately, the Viterbi search has no such communication pattern. This is why it must use level-pruning (a kind of beam pruning)[15] to reduce the communicate on requirements of the grammar expansion module in order to execute the resource management task in real time. It almost shows the opposite. Most systems use beam pruning to reduce the computation requirement of the word expansion module because the grammar expansion module is generally much less computation-loaded than the word expansion module. In ASPEN, the communication time is still about 40% of the total search time even with level-pruning [15]in comparison with 10% - 20% in other system. Therefore, the tree-machine is not suitable for either stack decoding or natural language processing which requires complicated synchronization (communication) and cannot take great advantage of massive parallelism (more than 10) in general. Because of the limited communication bandwidth and no shared memory scheme of ASPEN, the memory (mainly HMMs) distribution among PE's must be static. This makes dynamic load-balancing almost impossible on ASPEN when incorporating any kind of pruning strategy and therefore, the processor utilization would be poor. The training process which generally needs to access different HMMs during one-sentence training also does not fit in the tree-machine for the same reason.

Using more than 100 processors (or 1,000 MFLOPS) to achieve the DARPA resource management task in real time might be very expensive at first glance, but the great software and hardware scalability helps to reduce the development cost and time of ASPEN. Moreover, the binary tree architecture of ASPEN makes the pinouts of the PE's small and constant and thus makes ASPEN well-suited for miniaturization via VLSI and advanced packaging. It could also potentially reduce the cost of ASPEN.

4.4. MARS

MARS (Microprogrammable Accelerator for Rapid Simulation) [12] is a programmable pipeline multiprocessor machine which was originally designed for speed up VLSI circuit simulations. The MARS hardware is organized in three level: System, Cluster, and Processing Element (PE). The PE is a 16-bit custom designed microprogrammed VLSI processor with special features for message passing. Each PE has its own local memory ($64K * 16bits$). 15 PE's are then connected by a crossbar to form a cluster. The crossbar switch operates at 10 MHz and is dynamically configured at every clock cycle. A complete MARS system consists of up to 256 clusters connected by a binary n-cube communication network.

The implementation of Viterbi search on MARS is to partition the problem data space across the clusters (if necessary) and partition the algorithm over the PE's within a cluster. The local memory of each PE exclusively contains the data structures required by the part of algorithm executed in the PE. The PE's communicate by passing messages through a crossbar switch to form a pipeline. The algorithm partitioning is done carefully so that the processing load is evenly distributed among the PE's and the communication among them is minimized. For the benchmark test, they reported MARS can execute the resource management task in real time with a startup latency of about 25 μsec.

The use of pipeline in MARS to reduce the requirement of memory bandwidth is similar to SRI-Berkeley's search machine. However, the pipeline is programmable in MARS. This gives MARS some flexibility. For example, one can simply develop a continuous output distribution module and replace the discrete output distribution module in the pipeline with it (One might dedicate more PEs for a continuous output distribution module because of its complexity) in order to incorporate it with continuous densities. In contrast to the SRI-Berkeley's search machine, MARS is scalable because it can be expanded by adding more clusters.

Although a programmable pipeline structure is attractive, users needs to program very carefully to balance the load of every pipeline stage, so the slowest stage would not become the bottleneck of the entire system. However, no shared memory support would make such attempts more difficult because every PE can only access an equal limited amount of memory and some stages might need to access a large amount of memory. Since PE's in MARS are custom VLSI chips, users are forced to use MARS's microprogramming language which might prevent people from using MARS for development of speech algorithms. Moreover, since the clusters are connected by

a communication network, the long communication latency among clusters disallows MARS to perform complicated communication among clusters. Therefore, the partitioning of problem data space across the clusters must be static and this makes dynamic load balancing almost impossible. The long communication latency also make MARS unsuitable for algorithms that require frequent synchronization, like stack decoding and natural language processing. Even within the same cluster, MARS is still not suitable for algorithms that are hard to pipeline, like stack decoding and natural language processing. Although the training process can also be pipelined, the inability of PE's to access the whole data space would make MARS unsuitable for training.

MARS might not be as fast as the special purpose speech machines and it might not be easy to use for speech related algorithms. It still demonstrates how a flexible special purpose machine can be applied to other domains without re-design, though the machine is built for a different domain.

4.5. CMU's BEAM accelerator

CMU's BEAM [9] is a shared memory multi-processor hardware accelerator which is built on general-purpose processors. Three Weitek 8032 processors (10 mips) share an 8-Mbyte memory. Each of them also has a 32-Kbyte program memory and a 256-Kbyte local memory. The shared memory is the most popular architecture for parallel machines because of the visibility of the whole memory space, but they work only with algorithms with a high cache hit rate. The Viterbi algorithm reveals a poor cache hit rate and therefore requires a large memory bandwidth with respect to the number of instructions executed. BEAM use a dual-bank shared memory and local memories to increase memory bandwidth. The shared memory architecture also enables BEAM to perform synchronization effectively with the help of a *one-bit-per-word* flag which is managed directly by the hardware and is visible at the instruction level through *special-read* and *special-write* operations.

Due to the shared memory architecture, all processors can access all the HMM's. Therefore, the load balancing mechanism can be implemented very efficiently by using the synchronization flags without relocating the HMM's. This enables BEAM to incorporate beam search easily in comparison with the multi-processor architectures without shared memory. As a result, BEAM is able to execute the resource management task in real time with only three processors.

The big advantage of BEAM is the ease of development and the flexibility of the system. Since BEAM does not use any custom integrated circuits, it was developed quickly and cheaply.

In fact, the cost of BEAM is mainly bound to the cost of memory. In comparison with other architectures, BEAM is the cheapest in terms of the cost of architecture and development. The use of general-purpose processors also allows BEAM to take advantage of state-of-art processors and make BEAM easy to program (in the programming language C). In addition, the shared-memory structure makes BEAM feasible for almost all the speech applications. For example, BEAM will be ideal for training because of the ability of processors to access the whole memory space though the shared memory must be able to hold all the space required for training. Stack decoding and natural language processing can also be implemented on BEAM effectively because the hardware supported synchronization flag allows for complicated communication between processors.

However, like most other shared memory architectures, the memory bandwidth will be exhausted, even with a dual-bank shared memory when more processors are added to the system. Therefore, BEAM is not scalable. Although BEAM probably cannot handle a much larger task, it can indeed handle some tasks which require heavy computation with little change because of its flexibility. For example, to deal with continuous densities, one can add more processors (either the same processor or some floating point processors) which are dedicated to computing the output probabilities. The addition of those new processors will not result in memory contention because the memory for computing output probabilities could be totally separated from the memory describing the HMMs and therefore both parts can be executed simultaneously without interfering.

Although the speed of BEAM is still slower than the custom machines, like the SRI-Bekerley search machine, BEAM represents a good example of the trade-off between general purpose systems, which do not have the necessary speed, and custom systems, which take too much time and money to build. One might argue the special purpose hardware would be useless once the improvement of technology enables the general-purpose, single-processor machine to perform the task sufficiently. BEAM remains a good example of how general purpose technology can be used to build systems that are substantially faster than general purpose systems quickly and cheaply.

4.6. CMU's PLUS

Based on the pros and cons of the BEAM accelerator, CMU proposed a much more complicated multi-processor architecture, PLUS[10]. Shared memory is one of the most popular parallel processing models because of the simplicity of the programming model. However, bus-based shared-memory systems, like BEAM, do not perform well with a large number of processors

because of memory contention. Thus, PLUS attempts to use a physically distributed, but logically shared memory system to reduce the bandwidth requirements of a multiprocessor system. Difficulties with this approach include :

- Memory coherence in distributed memory systems is much more complicated than systems with a single memory.
- The communication latency results in intolerable delay for remote memory access and synchronization which could degrade the whole system performance.

Therefore, PLUS implements a software controlled, non-demand caching of replicated data to reduce remote access latency. The memory coherence is insured by a hardware supported coherence protocol mechanism (Coherence Manager). PLUS also supports a set of complex synchronization primitives and delayed synchronization techniques to reduce the overhead of synchronization.

The current implementation of PLUS uses a general purpose Motorola 88000 processor (25 MHz) with 32 Kbytes of cache and 8 or 32 Mbytes of two interleaved main memory at each node. Nodes are interconnected by a network implemented with Caltech's MOSIS router [16]. Each node can directly address the memory of any other node. Global memory mapping, coherence management are performed by a hardware module that is implemented with Xilinx PLD's and PAL's. PLUS can be connected to a workstation with either VME or NuBus. For the benchmark test, a two-node configuration can perform resource management task in real time.

From the users' points of view, PLUS inherits the superiority of BEAM, flexibility. Therefore, PLUS would be applicable for stack decoding, natural language processing and training because those applications require access to the whole problem space and need substantial synchronization in general. It is even possible to execute both speech recognition and natural language processing on the same PLUS machine and communicate with each other and therefore establish a complete spoken language system on a single PLUS machine. If the speech recognition component and the natural language component are implemented in different machines and interfaced by a local area network (LAN), the arbitrary delays between these two components would degrade the performance of the spoken language system in real applications. PLUS would prevent such arbitrary delays and allow the natural language component to closely control the search if necessary. On the other hand, its distributed memory structure enables it to execute programs locally and communicate through its shared memory scheme if necessary and prevent from saturating the memory bandwidth when

more nodes added to the PLUS system. Thus, the scalability of PLUS would be far better than BEAM, though the overhead of coherence management will still limit the range of scalability (less than 100). PLUS also supports a multi-threaded shared memory environment that is the same as Mach. This good programming environment enables users to develop and debug their programs on a single-processor or multiprocessor Mach host. This could substantially speed up the system development process.

Since PLUS is built on general purpose processors, it could take the advantage of the state-of-the-art general purpose processors. However, the custom integrated circuits and software to support complicated coherence management, data replication, synchronization primitives and delay synchronization would all increase the cost and efforts to build PLUS. Finally, the optimization of PLUS really depends on the data replication and delay synchronization. Data replication is very hard to optimize because the access pattern of application is unknown in general. One might use the access pattern of one test run to optimally re-allocate data space in subsequent runs, though the access pattern is very likely data-dependent. Like the delay branching issue in pipeline architecture, the power of delay synchronization relies highly on a good compiler; otherwise it would require careful low-level microprogramming skill of the users in order to take advantage of it.

PLUS represents a specific tradeoff in the space of distributed memory architectures that range from large-granularity, LAN-based machines, to message-passing machines, to bus-based shared-memory machines. Software development (including coherence management, data replication, delay synchronization and multi-threaded shared memory programming environment) play a very essential role in the real success of the system design. It indicate that better software development methodologies will have a substantial impact on the development time of such machines. It also reveals how software efforts could be incorporated into special purpose hardware design to further improve the performance of speech applications.

References

[1] Allen, J. *Cochlea Modeling.* **IEEE ASSP Magazine**, January 1985.

[2] Alleva, F. *Personal Communication.* unpublished, 1990.

[3] Bahl, L. R., Jelinek, F., and Mercer, R. *A Maximum Likelihood Approach to Continuous Speech Recognition.* **IEEE Transactions on Pattern Analysis and Machine Intelligence**, vol. PAMI-5 (1983), pp. 179–190.

[4] Bahl, L., Brown, P., De Souza, P., and Mercer, R. *Obtaining Candidate Words by Polling in a Large Vocabulary Speech Recognition System.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing.** 1988.

[5] Bahl, L., Brown, P., de Souza, P., and Mercer, R. a. *Estimating Hidden Markov Model Parameters so as to Maximize Speech Recognition Accuracy.* no. RC 13121 (#58589), IBM Thomas J. Watson Research Center, September 1987.

[6] Bahl, L. and et. al. *Large Vocabulary Natural Language Continuous Speech Recognition.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing.** 1989, pp. 465–467.

[7] Baker, J. K. *Stochastic Modeling as a Means of Automatic Speech Recognition.* Computer Science Department, Carnegie Mellon University, April 1975.

[8] Baum, L. E. *An Inequality and Associated Maximization Technique in Statistical Estimation of Probabilistic Functions of Markov Processes.* **Inequalities**, vol. 3 (1972), pp. 1–8.

[9] Bisiani, R., Anantharaman, T., and Butcher, L. *BEAM: An Accelerator for Speech Recognition.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing.** 1989, pp. 782–784.

[10] Bisiani, R. and M., R. *A distributed Shared-Memory System.* in: **International Symposium of Computer Architecture.** 1990.

[11] Brown, P. *The Acoustic-Modeling Problem in Automatic Speech Recognition.* Computer Science Department, Carnegie Mellon University, May 1987.

[12] Chatterjee, S. and Agrawal, P. *Connected Speech Recognition on a Multiple Processor Pipeline.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing.** 1989, pp. 774–777.

[13] Chow, Y. and et. al. *BYBLOS: The BBN Continuous Speech Recognition System.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing.** 1987, pp. 89–92.

[14] Cohen, J. *Application of an Auditory Model to Speech Recognition.* **The Journal of the Acoustical Society of America**, vol. 85 (1989), pp. 2623–2629.

[15] D., R., A., G., and P., R. *Incorporating Syntax Into The Level-Building Algorithm on a Tree-Structured Parallel Computer.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing.** 1989, pp. 778–781.

[16] Dally, W. and Seitz, C. *Deadlock-Free Message Routing in Multiprocessor Interconnection Networks.* **IEEE Transactions of Computers**, May 1987.

[17] Glinski, S., T.M., L., D.R., C., Koh, T., C., G., Wilson, G., and J., K. *A Tutorial on Hidden Markov Models and Selected Applications Speech Recognition.* **IEEE Proceedings**, vol. 75 (1987).

[18] Gorin, A. and D., R. *Parallel Level Building on a Tree Machine.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1988, pp. 295–298.

[19] Gorin, A. and R., S. *The ASPEN Parallel Computer, Speech Recognition and Parallel Dynamic Programming.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1987, pp. 976–979.

[20] Hennessy, J. L. *VLSI Processor Architecture.* **IEEE Transactions of Computers**, December 1984.

[21] Hon, H. and Lee, K. *On Vocabulary-Independent Speech Modeling.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1990.

[22] Huang, X., Alleva, F., Hayamizu, S., Hon, H., Hwang, M., and Lee, K. *Improved Hidden Markov Modeling for Speaker-Independent Continuous Speech Recognition.* in: **DARPA Speech and Language Workshop**. Morgan Kaufmann Publishers, San Mateo, CA, 1990, pp. 327–331.

[23] Jelinek, e. a. *A Real-Time, Isolated-Word, Speech Recognition System for Dictation Transcription.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1985.

[24] Lee, C., Rabiner, L., Pieraccini, R., and Wilpon, J. *Acoustic Modeling for Large Vocabulary Speech Recognition.* **Computer Speech and Language**, vol. 4 (1990).

[25] Lee, K. *Large-Vocabulary Speaker-Independent Continuous Speech Recognition: The SPHINX System.* Computer Science Department, Carnegie Mellon University, April 1988.

[26] Lee, K. and Hon, H. *Large-Vocabulary Speaker-Independent Continuous Speech Recognition.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1988.

[27] Lee, K., Hon, H., and Reddy, R. *An Overview of the SPHINX Speech Recognition System.* **IEEE Transactions on Acoustics, Speech, and Signal Processing**, January 1990, pp. 35–45.

[28] Leiserson, C. *Area-Efficient VLSI Computation.* Computer Science Department, Carnegie Mellon University, 1981.

[29] Linde, Y., Buzo, A., and Gray, R. *An Algorithm for Vector Quantizer Design.* **IEEE Transactions on Communication**, vol. COM-28 (1980), pp. 84–95.

[30] Lowerre, B. T. *The HARPY Speech Recognition System.* Computer Science Department, Carnegie Mellon University, April 1976.

[31] Lyon, R. *Analog Implementations of Auditory Models.* in: **Fourth DARPA Workshop Speech and Natural Language**. 1991.

[32] Lyon, R. *Computational Models of Neural Auditory Processing.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1984, pp. 36.1.1–4.

[33] Markel, J. D. and Gray, A. H. **Linear Prediction of Speech**. Springer-Verlag, Berlin, 1976.

[34] Murveit, H. and Weintraub, M. *Speaker-Independent Connected-Speech Recognition Using Hidden Markov Models.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1988.

[35] Ney, H. *Dynamic Programming Speech Recognition Using a Context-Free Grammar.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**, IEEE ASSP. 1987, pp. 3.2.1 – 3.2.4.

[36] Ney, H., Mergel, D., Noll, A., and Paeseler, A. *A Data-Driven Organization of the Dynamic Programming Beam Search for Continuous Speech Recognition.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1987, pp. 833–836.

[37] Nilsson, N. **Principles of Artificial Intelligence**. Tioga Publishing Co., Palo Alto, CA, 1980.

[38] Nilsson, N. **Problem-Solving Methods in Artificial Intelligence**. McGraw-Hill, New York, 1971.

[39] Patterson, D. A. *RISC-I : A reduced instruction set VLSI computer.* in: **8th. Annu. Symp. Computer Architecture**. 1981.

[40] PAUL, D. *Personal Communication.* 1990.

[41] Paul, D. *The CSR-NL Interface Specification.* in: **DARPA Speech and Language Workshop**. 1989.

[42] Paul, D. *The Lincoln Robust Continuous Speech Recognizer.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1989, pp. 449 – 452.

[43] Paul, D. and Martin, E. *Speaker Stress-Resistant Continuous Speech Recognition.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1988.

[44] Price, P., Fisher, W., Bernstein, J., and Pallett, D. *A Database for Continuous Speech Recognition in a 1000-Word Domain.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1988, pp. 651–654.

[45] Rabaey, J., Broderson, R., Stoelzle, A., Chen, D., Narayanaswamy, S., Yu, R., Schrupp, P., Murveit, H., and Santos, A. *A Large-Vocabulary Real-Time Continuous-Speech Recognition System.* in: **VLSI Signal Processing, III**. IEEE Press, New York, NY, 1988.

[46] Rabiner, L. R. *A Tutorial on Hidden Markov Models and Selected Applications Speech Recognition.* **IEEE Proceedings**, 1988.

[47] Rabiner, L. R., Juang, B. H., Levinson, S. E., and Sondhi, M. M. *Recognition of Isolated Digits Using Hidden Markov Models With Continuous Mixture Densities.* **AT&T Technical Journal**, vol. 64 (1985), pp. 1211–33.

[48] Rabiner, L., Wilpon, J., and Soong, F. *High Performance Connected Digit Recognition Using Hidden Markov Models.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1988.

[49] Schwartz, R., Chow, Y., Kimball, O., Roucos, S., Krasner, M., and Makhoul, J. *Context-Dependent Modeling for Acoustic-Phonetic Recognition of Continuous Speech.* in: **IEEE International Conference on Acoustics, Speech, and Signal Processing**. 1985, pp. 1205–1208.

[50] Viterbi, A. J. *Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm.* **IEEE Transactions on Information Theory**, vol. IT-13 (1967), pp. 260–269.

[51] Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. *Phoneme Recognition using Time-Delay Neural Networks.* **IEEE Transactions on Acoustics, Speech, and Signal Processing**, vol. ASSP-28 (1989), pp. 357–366.

[52] Zue, V., Glass, J., Phillips, M., and Seneff, S. *The MIT SUMMIT System: A Progress Report.* in: **Proceedings of DARPA Speech and Natural Language Workshop**. 1989.