ABSTRACT

BAPAT, OJAS ASHOK. A Generic, Scalable Architecture for a Large Acoustic Model and Large Vocabulary Speech Recognition Accelerator. (Under the direction of Paul Franzon.)

This dissertation describes a scalable hardware accelerator for Speech Recognition. We propose a generic hardware architecture which can be used with multiple software which use HMM based Speech Recognition . We implement a two pass decoding algorithm with an approximate N-best time synchronous Viterbi Beam Search. The Observation Probability Calculation (Senone Scoring) and first pass of decoding, which uses a simple language model, is implemented in hardware. A word lattice, which is the output from this first pass, is used by the software for the second pass, with a more sophisticated N-gram language model. This allows us to use a very large and generic language model in our hardware. We opt for the logic-on-memory approach to make use of a high bandwidth NOR Flash Memory to improve our random read performance for senone scoring and first pass decoding, both of which are memory intensive operations. For senone scoring, we store all of the acoustic model data in NOR Flash Memory. For the decoding, we partition the data accesses between DRAM, SRAM and NOR Flash, which allows parallelism of these accesses and improves performance. We arrange our data structures in a specific manner, which allows complete sequential access of the DRAM, thereby improving memory access efficiency. We use techniques like block scoring and caching of word an HMM models to reduce the overall power consumption and further improve performance. The use of a word lattice to communicate between hardware and software keeps the communication overhead low, compared to any other partitioning scheme. This architecture provides us with a speed up of 4.3X over a 2.4 GHz Intel Core 2 Duo proces-

sor running the CMU Sphinx recognition software, while consuming an estimated 1.72 W of power. The hardware accelerator provides improved speech recognition accuracy by supporting larger acoustic models and word dictionaries while maintaining real time performance.

A Generic, Scalable Architecture for a Large Acoustic Model and Large Vocabulary Speech Recognition Accelerator

by
Ojas Ashok Bapat

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2012

APPROVED BY:

W. Rhett Davis

Eric Rotenberg

Robert Rodman

Paul Franzon
Chair of Advisory Committee

DEDICATION

To my Loving Family

BIOGRAPHY

Ojas Bapat is a PhD Candidate at the Electrical and Computer Engineering Department of North Carolina State University, Raleigh, NC. He currently works as a Systems Hardware Engineer for Spansion Inc. in Sunnyvale, CA. He earned his Master's Degree from NC State University in 2009 under the guidance of Dr. Paul D. Franzon. He has worked on a variety of topics including Speech Recognition, in-situ calibration of circuits using an on-chip co-processor and DDR2 controller design. His current research interests include Speech Recognition Algorithms, Development of HW/SW Co-Designs and Co-Processors, Hardware performance modeling and design using SystemC/C++ and RTL.

ACKNOWLEDGEMENTS

I thank my advisor, Dr. Paul D. Franzon, for giving me the opportunity to work with him. His advice has been of great help. My knowledge as well as interest towards ASIC design has strengthened only because of his extremely effective ways of teaching in class.

I would also like to thank Dr. W. Rhett Davis, Dr. Eric Rotenberg and Dr. Robert Rodman for serving on my advisory committee and reviewing my research work.

I specially thank Spansion LLC., Sunnyvale, CA. This work was carried out under research contract 5582374 between NC State University and Spansion.

I thank Richard Fastow from Spansion, LLC for his guidance while I was working on this topic on site in Sunnyvale, CA.

I thank my wife Neeti for sharing joys and sorrows and for being a good listener, punching bag and much much more.

I would also like to thank all my friends from Gorman St., without whom, the six years spent at NCSU wouldn't have been so exciting.

Above all, I thank my parents for their unconditional support in my endeavors. They have always been there for me. They give me the motivation and help me rise every time I fall.

TABLE OF CONTENTS

List of Tables . . . . **viii**

List of Figures . . . . **ix**

Chapter 1 Introduction **1**
1.1 Key Contributions 4
1.2 Dissertation Outline 8

Chapter 2 Speech Recognition Theory **10**
2.1 Parameters in Speech Recognition Systems 10
2.1.1 Recognition Rate 10
2.1.2 Accuracy 11
2.1.3 Word Error Rate 11
2.2 DSP Front End 11
2.3 Hidden Markov Models 12
2.4 Neural Networks 12
2.5 Acoustic Modeling using HMMs 12
2.6 Word Dictionary 14
2.7 Stochastic Language Modeling in Speech Recognition 14
2.7.1 N-gram Language Models 15
2.8 Weighted Finite State Transducers 15
2.9 Viterbi Decoding 16
2.10 N best Search 17
2.11 N best Search Approximations 18
2.11.1 Lattice N best 18
2.11.2 Word Dependent N best 18
2.12 Multi Pass Decoding 19
2.13 Fast Match 20
2.14 Word Lattice 20
2.15 Stack Decoding 21

Chapter 3 Related Work **23**
3.1 Existing Software Solutions 23
3.2 Existing Hardware Solutions 24
3.3 This Work 28

Chapter 4 Proposed Hardware Architecture **29**
4.1 Design Space Exploration 29
4.2 Overall System Architecture 41
4.3 Hardware Software Partition 41
4.4 Software Commands 42
4.5 Hardware Output 46
4.6 Senone Score Unit 47
4.6.1 Block Senone Scoring 48
4.6.2 Flash Control and Memory Structure 50
4.6.3 Distance Calculation 50
4.6.4 Logarithmic Addition 50
4.7 Viterbi Decode Unit 52
4.7.1 Flash Control and Memory 53
4.7.2 HMM scoring 53
4.7.3 Adaptive Pruning 53
4.7.4 Active List Generation 58
4.7.5 Word Lattice Generation 59
4.7.6 New Word and HMM activation 59
4.8 Scalability of the Design 60

Chapter 5 Hardware Modeling Methodology **63**
5.1 System C 63
5.2 Transaction Level Modeling (TLM 2.0) 64
5.2.1 Loosely Timed Models (LT) 64
5.2.2 Approximately Timed Models (AT) 64
5.2.3 Mixed Model 65
5.3 TLM vs RTL 65
5.4 Interaction between SystemC Model and CMU Sphinx 66
5.5 Structure of the SystemC Model 67
5.6 Transactions, Transport calls and Event Generation 68
5.7 SystemC Behavioral Model of the Hardware 69
5.7.1 Utterance Initialization 69
5.7.2 Frame Initialization 69
5.7.3 Flash Control for SSU 70
5.7.4 Distance Calculation 71
5.7.5 Adding Mixture Weight and changing Log Base 71
5.7.6 Logarithmic Addition 72
5.7.7 Memory Model 72
5.7.8 Viterbi Decoder Top Level Function 73
5.7.9 Flash Control for VU 74
5.7.10 DRAM Controller 75

5.7.11 Next Word / HMM Activation Block 75
5.7.12 Fetching Senone Scores for Triphone Scoring 76
5.7.13 Scoring each phone in a triphone 76
5.7.14 HMM Pruning 77
5.7.15 HMM Propagation 77
5.7.16 Word Pruning based on word threshold 78
5.7.17 Adding recognized words to the output lattice 79
5.7.18 Calculation of Adaptive Pruning Threshold 79
5.7.19 SPI interface specific functionality 80
5.7.20 Interpretation of commands from CPU 81
5.7.21 Top level wire connections and signals 81
5.8 Estimation of Timing Parameters for the SystemC Model 81
5.9 DRAM performance estimation 82
5.10 Architectural Exploration using SystemC Model 83
5.11 Area and Power Estimation 86
5.11.1 DRAM Power Estimation 86
5.12 Communication Overhead Modeling 89
5.12.1 Serial Peripheral Interface (SPI) 89
5.13 Validation of SystemC Model 90
5.14 Word Error Rate Calculation 91

Chapter 6 Results **92**
6.1 Accuracy 92
6.2 Estimated Performance, Area and Power 94
6.3 Comparison to other work 96

Chapter 7 Conclusion **98**
7.1 Summary 98
7.2 Future Work 99

References **101**

LIST OF TABLES

Table 4.1 Hardware Software Partitions Investigated 30
Table 4.2 Communication Bandwidth Requirement for various HW/SW Splits 32
Table 4.3 Data accessed every frame for senone scoring and decode pass I . . 38
Table 4.4 Memory Access Efficiency for different Hardware configurations . . 39

Table 5.1 Simulation time for SystemC TLM Models and RTL for 1 sec of real time speech 65

Table 6.1 Effect of Gaussian Mixture on WER using Multipass Decode . . . 93
Table 6.2 WER with N-best Bigram Pass I and 1-best Trigram Pass II . . . 93
Table 6.3 Power consumption itemized by task 95
Table 6.4 Memory Access Efficiency for different Hardware configurations . . 95
Table 6.5 SSU Comparison with Related Work 97
Table 6.6 VU Comparison with Related Work 97

LIST OF FIGURES

Figure 1.1 An HMM based Speech Recognition System 2
Figure 1.2 Hardware Software Partition for Speech Recognition using Multi-pass Decoding 5

Figure 2.1 Hidden Markov Model for a Tri-Phone 13
Figure 2.2 Dictionary entry for the word started 14
Figure 2.3 Viterbi Search Transitions through HMMs 17
Figure 2.4 Example of a Word Lattice 21

Figure 4.1 Performance comparison with sphinx 3.0 running on desktop PC with various HW/SW splits for acceleration 31
Figure 4.2 Effect of Number of Gaussian Mixtures in the Acoustic Model on Word Error Rate 33
Figure 4.3 Word Error Rate for different language models in Pass I and Tri-gram in Pass II 34
Figure 4.4 Hardware architechtures explored I 35
Figure 4.5 Hardware architechtures explored II 36
Figure 4.6 Comparison between hardware accelerator with and without feed-back for senone activation 36
Figure 4.7 Number of Senones Active for each frame for 5K dictionary with narrow beam width and 64K dictionary with wide beam width . . 37
Figure 4.8 HMM cache Hit rate for various cache sizes and values of N (number of hypothesis kept active at every word) 40
Figure 4.9 Top Level Hardware Block Diagram 42
Figure 4.10 Word Lattice Output of First Decode Pass (Hardware) 47
Figure 4.11 Block Diagram for a Scalable Senone Score Unit 47
Figure 4.12 Pipelining of operations of the Speech Recognition Algorithm . . . 48
Figure 4.13 SSU Power Consumption for various block sizes for block senone scoring 49
Figure 4.14 Packed Data Stucture for Acoustic Models in NOR Flash 51
Figure 4.15 Block Diagram for a Scalable Viterbi Unit 52
Figure 4.16 Flash Read Control for the Viterbi Unit 54
Figure 4.17 Flash Memory Format for the Viterbi Unit 55
Figure 4.18 Storage Format for each Word and HMM 56
Figure 4.19 Phone Scoring Unit 56
Figure 4.20 Number of Active HMMs per frame over an entire utterance which lasts 381 frames 57
Figure 4.21 Active List Format in DRAM 58

Figure 4.22 New Word and HMM Activation Block 59
Figure 4.23 Word Activation Map 61
Figure 4.24 Hardware Scalability 62

Figure 5.1 Example Simulation with SystemC 66
Figure 5.2 Structure of the SystemC Model 67
Figure 5.3 DRAMsim2 Usage 83
Figure 5.4 SystemC Simulation to explore effect of parallelism and pipelining on overall SSU performance 85
Figure 5.5 Counters for Power Estimation in the Hardware SystemC Model . 87

Figure 6.1 Time taken by each operation in Multi-Pass Decode on Intel Core2Duo 2.4 GHz with 4GB 667MHz DDR2 SDRAM; with and without the Proposed Hardware Accelerator (400 utterances, WSJ0 CSR I corpus) 96

Chapter 1

Introduction

Automated Speech Recognition is increasingly becoming the preferred mode of interaction with computers today, especially in mobile devices and automobiles. It also has extensive applications in medical transcriptions, closed captioning for live events, etc. Speech is the most natural method of communication for humans.

The ultimate goal for a speech recognition system is high accuracy, large vocabulary, low latency, speaker independence and immunity to background noise. Such systems need to handle more than 60-70K words and should be able to recognize natural speech. These systems need a modern desktop machine to give a result in real time. To use these systems on mobile phones or other equipment which does not possess the required computing requirements, a trade-off has to be made with accuracy, in order to achieve real time performance.

There is need to develop efficient implementations of the speech recognition algorithms for various purposes. One is to facilitate the use of high accuracy speech recognition in embedded systems like cell phones and automobiles, while the other is to lower power consumption in server based speech recognition.

Use of Application Specific Integrated Circuits (ASICs) has been known to provide magnitudes of improvements in performance and power over general purpose processors. One drawback of ASICs is the lack of flexibility, compared to software solutions. Hence, we choose a hardware software co-design approach which can provide the performance of an ASIC and the flexibility of software.

Shown in Fig. 1.1 is a generic Hidden Markov Model (HMM) based speech recognition system. It consists of three main stages, the DSP front end, the acoustic modeling and the language modeling. The front end stage converts the incoming speech to an energy spectrum which is represented using a feature vector. This feature vector is matched against the pre stored models of sounds (phones), by the acoustic modeling stage. This gives use the most probable sounds for the incoming speech. The decoding stage finds out the best sequence of sounds and the best sequence of words corresponding to them, using the word dictionary and language models.

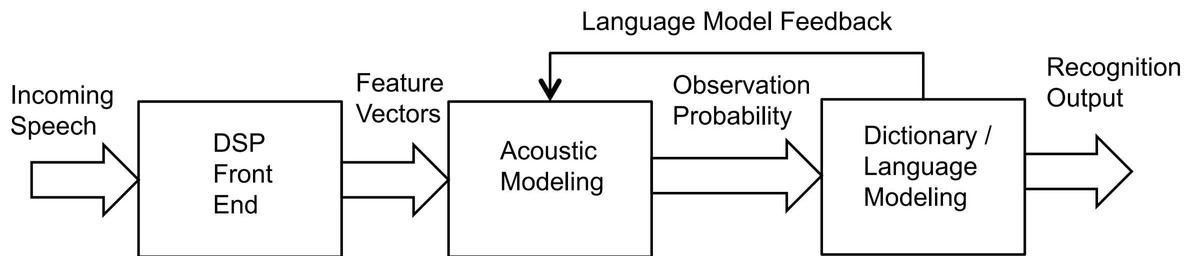


Figure 1.1: An HMM based Speech Recognition System

The major factors limiting the accuracy of speech recognition are high computational and memory bandwidth requirement. Many have chosen to tackle these problems in different ways. [1, 2, 3] show that memory bandwidth and cache pollution greatly hamper performance. [4, 5] show that the memory access patterns for the search phase of the

algorithm are random and sporadic in nature. They also propose one method to optimize these accesses. A common method to achieve memory bandwidth reduction is by reducing the precision of acoustic and language models used in the recognition procees. [4, 5] show that it is possible to maintain accuracy using 8,16 bit data while [1, 6, 2] use 24 bit data. We choose to solve the computational bottleneck by using parallelism in hardware. For the memory bandwidth bottleneck, we use a high bandwidth on chip NOR Flash memory and many accompanying algorithmic optimizations.

The acoustic models used in speech recognition model sounds in a language as a set of gaussian mixtures [7]. The number of mixtures used here has an effect on the end accuracy of recognition. A very large number of mixtures is required to represent some of the sounds in the language adequately. This implies more computation and memory reads. The language models used for attaching word probabilities to the search have a huge impact on the recognition process [8]. However, they do also add complexity and introduce the need to back track during the search process. It becomes increasingly difficult to have hardware which provides guaranteed performance for any type of language model.

The most commonly used algorithm for the decode phase of speech recognition is the viterbi beam search [9]. This dynamic programming algorithm makes one approximation, i.e. it follows (continues with) only the best incoming path into a node. For this technique to always find the ultimate best path, it is essential that the dynamic programming invariant is not violated. Sophisticated N-gram language models [8] violate this condition by definition by assigning weights to sequence of more than two words. The way around this is to follow multiple best paths coming into a node during the search. Such algorithms are called N-best search algorithms [10, 11], and we use one of these algorithms in our hardware implementation. These algorithms have high complexity but provide

benefits like improved locality of memory accesses and use of a simple language model, if the ultimate goal is to present multiple hypothesis and not the selection of a single best hypothesis. From the results we achieved, we will see that these benefits compensate for the increased complexity of the algorithm and provide better performance when implemented in hardware.

Keeping all these factors in mind, we split the entire speech recognition process into four stages, as shown in Fig. 1.2. The decode process is split into two stages. This technique is known as multipass decoding[12]. In the first stage, we try to find the top N best possible sequences using an approximate word dependent N-best [11] time synchronous viterbi beam search on a unigram/bigram language model. The N-best sequences are represented in the form of a word lattice . The second pass of decode finds the best hypothesis from this word lattice. Observation Probability Calculation and N-best Search are performed in hardware. The DSP front end is still implemented in software, since it is not a sophisticated front end and does not require much computing power. The second stage of decode which uses 1-best search on a reduced search space (word lattice) is implemented in software. Since the second stage works on a very limited search space, it allows of use of higher N-grams or possibly target application specific N-grams.

1.1 Key Contributions

- Design Space Exploration: We explore the affect of various hardware / software splits on the performance of the speech recognition system. Improved performance can be translated to improved accuracy by supporting larger acoustic models and word dictionaries. We also explore the communication bandwidth requirements for

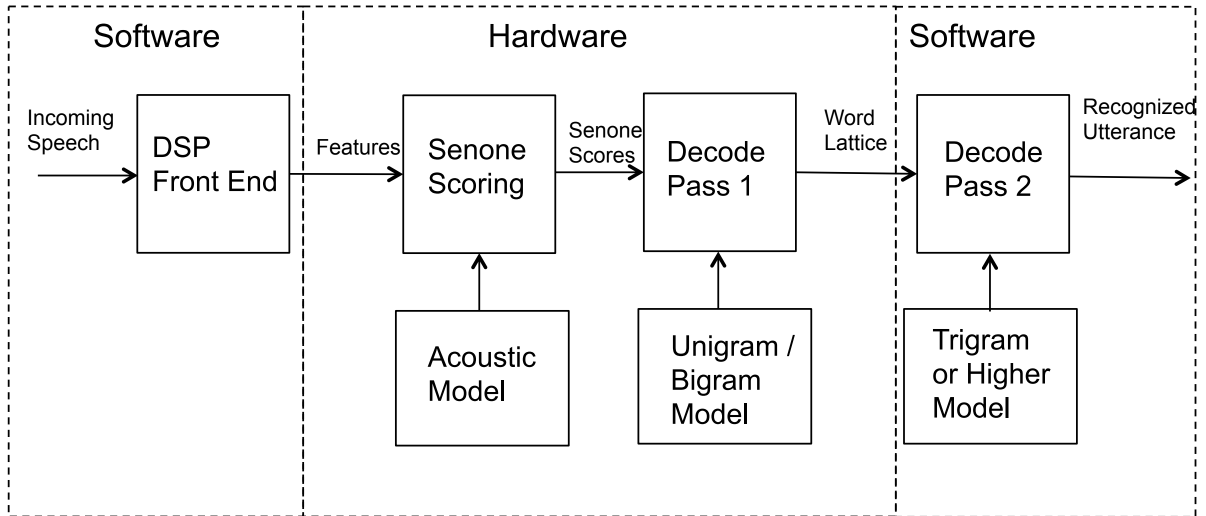


Figure 1.2: Hardware Software Partition for Speech Recognition using Multipass Decoding

different HW/SW configurations.

- Hardware / Software System Partitioning Scheme: We propose a hardware software co-design which implements a two pass decoding algorithm. Observation Probability calculation (senone scoring) and the first pass of decode is implemented in hardware. The First pass of decode performs a coarse search using a simple Unigram/Bigram Language Model. Use of a simple language model reduces search complexity as the model is context dependent only on the previous word and does not require backtracking to find the best path. This largely simplifies hardware design. The word lattice format used as output of first pass contains acoustic scores for all words and also path information which can be easily converted into other N-best notations, for use in the further stages of decoding.

- Generic and Scalable Architecture: We propose an architecture that can easily scale for use with acoustic model size, dictionary size, number of states in an HMM,

number of Gaussian mixtures in a senone and dimensionality of the feature vector. The input of the hardware in feature vector and output is a word lattice with score, time and path information, which makes this usable with any software which does Hidden Markov Model / Statistical Language Model based speech recognition. It can also be scaled for server applications where it can support multiple input streams simultaneously.

- Hardware Implementation of a word dependent N-best algorithm: We implement the word dependent approximate N-best algorithm [10, 11] in hardware, for the first path of decode. This algorithm has less complexity compared to other N-best algorithms. Since we use a Unigram/Bigram language model, this algorithm suits our needs as it follows an N-best path only at a single word level. We also use adaptive pruning [13] to check the size of our N-best list. This removes the sorting requirement for keeping the top N-best paths.

- Partitioning of volatile and non-volatile data: We found that for Observation Probability Calculation, most of the data accesses is non-volatile. We solved the memory bottleneck for this phase by using an on-chip high bandwidth NOR Flash Memory. For the decode phase, the accesses to Word and HMM dictionaries are inherently random and non-volatile. Again, a high bandwidth NOR Flash memory suits our purpose. We store word models as a sequence of HMM IDs. The HMM models are stored separately. Since HMM structures are re-used in multiple words, this reduces memory size requirement.

- Sequential Active List access scheme: The single most factor which helps us achieve improved performance is the fact that we access our entire active list in DRAM sequentially. We store the active list in the order of increasing word IDs. Multiple

(N-best) instances for a single word are all clustered together. This allows us to read the word model for that word from the flash only once, and re-use it for all the word instances. The clustering or words in the word dictionary by their starting senones allow for insertion of new words into the active list, without breaking the sequential access pattern.

- New Word and HMM activation technique: We perform the activation of new HMMs within a word as soon as we read the word from the active list in DRAM. This activation is performed based on scores stored in the last frame. This reduces DRAM bandwidth requirement compared to the other approach which activates HMMs before writing them to the DRAM. For new word activation, we use a word activation map which maps the words in the lattice to the starting senones of other words in the language model. This allows us to easily insert or modify N-best word instances in the active list, on the fly. Activation of new words based on word lattice entries from the last frame reduces DRAM bandwidth as many of the new words inserted get pruned out before begin written to the DRAM for the next frame.

- Caching of HMM model data for a single word: Since we cluster all instances of a word in the active list, an obvious extension would be leverage the temporal locality of this data pattern. We use an SRAM buffer big enough to store all HMMs in a single word. We read the word model only once and use the HMM IDs to access the HMM structures. We read the HMM structures for the active HMMs from NOR flash and store them in the SRAM buffer. So, for other instances of the same word, these HMMs can be read from the SRAM buffer. This improves performance and reduces power consumption.

- Prefetching of Bigram model data for word lattice entries: We index bigram prob-

abilities in the Flash memory by the source word ID. For word lattice entries added in a frame, we prefetch multiple lines of bigrams for all these words into an SRAM,which has equal space allocated for bigrams of each source word. Fetching bigrams by source word ID reduces the total memory bandwidth. Bigrams are accessed from the prefetch SRAM in the order of destination IDs, i.e. in the same sequence as the active list.

- Parallel Operation of all stages in the Speech Recognition Algorithm : In our design space exploration, we realized that for wide beamwidth (larger than 10%) large vocabulary (larger than 64K words) speech recognizer, about 92% of the acoustic model is always active. Hence, we remove the feedback from the decode stage back to the acoustic stage and score the entire acoustic model for all frames. This allows all the stages in the speech recognition algorithm to work in parallel.

- Block Senone Scoring: Removal of feedback from decode stage back to the acoustic modeling stage implies that the observation probability calculation now depends only on the incoming feature vector. We leverage this by calculating observation probabilities over multiple frames. Doing this reduces both the average and peak memory bandwidth requirement and greatly reduces the power consumption.

1.2 Dissertation Outline

In chapter 1 we introduced you to some speech recognition basics, problems and our motivation for this work. We also outlined our key contributions. In chapter 2 we will dig deep into the some theory on how the recognition algorithm actually works and the mathematics involded. Next, in chapter 3, we discuss prior work done in this field and

industry state of art.In chapter 4, we will dive into our proposed hardware architecture and discuss each of the block in detail. In this chapter, we will also describe how we explored our design space and arrived at the optimal design point. Since we model the entire architecture in SystemC, it is of utmost importance to discuss this. We do this in chapter 5, where we look at our modeling methodology, area and power estimation process and DRAM modeling process. Then we show some key results and comparison with other work in chapter 6. We make concluding remarks in chapter 7 and suggest some future work.

Chapter 2

Speech Recognition Theory

In this chapter, we will discuss the algorithms used for speech recognition and the parameters used for evaluation of speech recognition systems. The most widely used algorithms in speech recognition use Hidden Markov Models (HMMs). These are the ones we will concentrate on, as they are greatly benefited by hardware acceleration. We will also briefly discuss CMU Sphinx, which was used as the software for our proposed hardware-software co-design.

2.1 Parameters in Speech Recognition Systems

Let us first look at some of the parameters and terms in speech recognition systems.

2.1.1 Recognition Rate

It is the length of time it takes a recognition system to process a speech utterance compared to the length of the utterance itself, as shown in equation 2.1.

$$R.R. = \frac{T_{recognition}}{T_{utterance}} \tag{2.1}$$

2.1.2 Accuracy

Accuracy of a speech recognizer is the percentage of words it identifies correctly. For a speech sample of N words, accuracy is defined in equation 2.2.

$$WER = \frac{N_{correct}}{N} \tag{2.2}$$

2.1.3 Word Error Rate

Word Error Rate is the total number of erroneous words compared to the correct hypothesis. Hence, it considers words that get substituted, inserted or deleted. For a speech sample of N words, word error rate is defined in equation 2.3.

$$WER = \frac{N_{sub} + N_{ins} + N_{del}}{N} \tag{2.3}$$

2.2 DSP Front End

The goal of a speech recognition system is to recognize the uttered sequence of words. At the front end, input speech is sampled and a spectral analysis is performed to generate feature vectors to represent this speech. These feature vectors are generated at set intervals called frames. Each such feature vector is called an observation. The duration of a frame depends on the front end. CMU Sphinx uses 10ms frames. Thus, the input speech is now converted to an observation sequence. A detailed explanation of the working of the DSP front end can be found in [12] and the documentation for CMU Sphinx 3.0 [14].

2.3 Hidden Markov Models

Hidden Markov models are a variation of regular Markov models and are useful when the underlying state model of a system is unknown. The concept is similar to markov chains but the observation is now a probabilistic function of the state. The result is a doubly embedded stochastic process with an underlying process which is not observable. It can be observed only through another set of stochastic processes that produce a sequence of observations. A detailed explanation of Markov chains and HMMs can be found in [15]. The application of HMMs to speech recognition is discussed in depth in section 2.5.

2.4 Neural Networks

Neural networks can be used in speech recognition to represent observation probabilities, similar to HMMs [16]. Most Neural network models today are found in small to medium vocabulary, continuous recognition tasks. There are also some hybrid neural network / HMM techniques.

2.5 Acoustic Modeling using HMMs

Every spoken word in the language is represented in terms of basic sounds called phones. The pronunciation of every phone is affected by its context, i.e. the phones preceding it and succeeding it. Thus, the phones are clubbed with their neighbouring phones to form a context dependent units called triphones. The total number of possible triphones can be very large for any language, e.g. there are 50 phones in english language and a possible 50^3 triphones. Each state in the triphone represents a phone. There are various HMM structures that can be used to model context dependent phone units [15]. We

choose a simple left to right statistical hidden markov model (HMM) as shown in Fig. 2.1. The transitions between states of the HMM have probabilities attached to them. These probabilities model time duration for the HMM. A collection of such context dependent HMMs is called the acoustic model. The emission probabilities for each state in the model are represented as gaussian mixtures. During the training of the acoustic models [15], the means and variances for the mixtures are calculated and stored. States in the model which have similar gaussian distributions are clustered together into units called senones [7]. This greatly reduces training effort. During the process of speech recognition, the mahanalobis distance is calculated between the incoming features of speech and the trained mixiture data for each senone. The result is the observation probability of each senone, as shown in equation 2.4. This process is called gaussian estimation. In all hardware and software implementations of speech recognition systems today, this equation is calculated in logarithmic domain, to reduce the multiplications and remove the exponent. This does add the necessity for a logarithmic addition, but this can be achieved easily using a look up table approach.

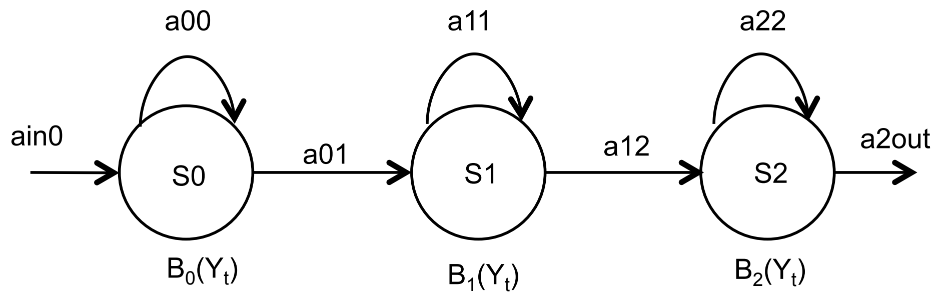


Figure 2.1: Hidden Markov Model for a Tri-Phone

$$b_j(Y_t) = \sum_{m=1}^{M} C_{jm} \prod_{n=1}^{N} e^{(Y_t[n]-\mu_{jm}[n])^2 * V_{jm}[n]} \tag{2.4}$$

$$logb_j(Y_t) = \sum_{m=1}^{M} C_{jm} \sum_{n=1}^{N} (Y_t[n] - \mu_{jm}[n])^2 * V_{jm}[n] \tag{2.5}$$

2.6 Word Dictionary

A word dictionary used in speech recognition models each word in terms of the basic HMM units that have been used in the acoustic model. If we use triphones, each word gets modelled as a sequence of triphones. If we use context independent phones in our acoustic model, the word will be modelled as a sequence of phones as shown in Fig. 2.2.

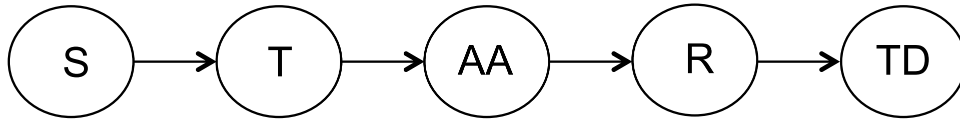


Figure 2.2: Dictionary entry for the word started

2.7 Stochastic Language Modeling in Speech Recognition

A language model is used for defining the semantics or grammar for the language. It models the probabilities of occurrence of different words in the dictionary. The Viterbi search algorithm used in speech recognition finds the most probable sequence of phones, for an observed utterance. The word dictionary and the language model together help

us find the most probable sequence of words from this sequence of phones. A stochastic language model (SLM) attaches probabilities to individual words or a sequence of words. Certain sequences have higher probability compared to other sequences. This helps us choose one set of words over another, even if they map to the same set of phones.

2.7.1 N-gram Language Models

N-gram models are called so because they attach probabilities to sequences of N words. The simplest N-gram model is a Unigram, which deals with individual words. A Bigram would attach probabilities to a sequence of two words, a trigram to three words and so on. Bigram or trigram models do not store the the probabilities for every word pair or triple as this would be impractical. A 64K dictionary would have 262 Trillion trigrams. Hence, only the most probable bigrams or trigrams are stored. We need to use some hashing or indexing technique to find out if a bigram or trigram exists for a given word sequence. All N-gram models use a fall back strategy. For example, if a trigram does not exist, we have to fall back on the bigram, and then to a unigram [17].

2.8 Weighted Finite State Transducers

Finite state transducers [18] are another method of representing language contexts in speech recognition. The entire grammar is represented in the form of a lexical tree which is traversed from root to leaves during the decoding process [1]. This is a very efficient way of storing the language models and is much faster to traverse compared to a traditional statistical language model. However, the main drawback of this approach is that it cannot handle word sequences that are outside the grammar.

2.9 Viterbi Decoding

The Viterbi Decoding Algorithm [9] finds the most likely sequence of phones for an observed utterance. With the use of the language model, this sequence is mapped to a sequence of words. Equation 2.6 shows the probability for a sequence of N words. $P(W)$ is the language model probability and $P(O|W)$ is the observed probability.

$$W_1.....W_n = arg_w max \frac{P(W).P(O|W)}{P(O)} \tag{2.6}$$

Every word can be thought of as one long HMM comprising of several small HMMs like triphones. Equation 2.7 is used to calculate the probability that the HMM will be in a particular state. It considers only the best forward path coming into the state, for the probability calculation. The weight of the transition path is added to the probability and so is the acoustic probability of the state itself. An example Viterbi trellis is shown in Fig. 2.3. The decoder starts at state q_0 and moves through the HMM states with each observation. For each observation t, the state scores are calculated using the scores from frame $t-1$.

$$log(\delta_t(j)) = \max_{1 \leq i \leq N} [log(\delta_{t-1}(i)) + loga_{ij}] + logb_j(Y_t) \tag{2.7}$$

It is however impractical and unnecessary to calculate the probabilities for all the states in the language model for every frame. So the search is bounded by a threshold and all states below that threshold are pruned out for further observations. This strategy is called Viterbi Beam Search [19] is used in most speech recognizers today.

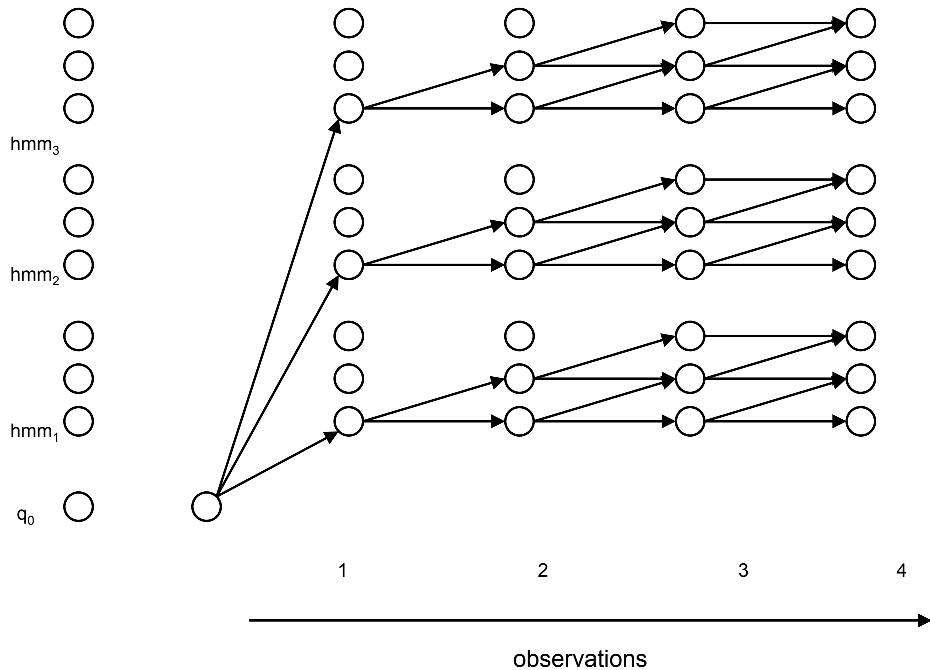


Figure 2.3: Viterbi Search Transitions through HMMs

2.10 N best Search

The optimal N-best decoding algorithm [10] is quite similar to the Viterbi decoder. However, N-best search finds all hypothesis sequences within a specified beam width. It keeps track of different hypotheses by maintaining different histories at each state. It then passes the top N hypothesis to the next stage of the search. However, in its pure form it can be partial to shorter hypotheses. Also, most of the hypotheses picked in the first stage differ only marginally, and therefore lot of results get duplicated. This is overcome by using some approximations discussed below. The aim of the approximations is to reduce the computation as much as possible, while keeping the best hypothesis in the list.

2.11 N best Search Approximations

Approximations of the N best algorithm allow us to reduce the computations while guaranteeing that the best hypothesis is present in the list. Discussed below are two such algorithms.

2.11.1 Lattice N best

This algorithm uses a forward pass time synchronous search within words [11] while storing only one path for each state. At each word boundary, instead of remembering the best scoring word, all the different words with their scores are saved for backtracking later. This requires no extra computation above the 1-best algorithm. The problem with this algorithm is that it can miss high scoring hypothesis [11] since it allows only one score to be stored at states within a word.

2.11.2 Word Dependent N best

The word dependent N best algorithm [11] is a compromise between exact N best and lattice N best algorithms. It uses the fact that the best starting point of a word is dependent only on the word preceding it. Hence, at each state within a word, scores corresponding to paths from all preceding words are stored separately. At the end of the word, a score is recorded for each combination of that word and the preceding word. Only single path is extended forward from the word. The computation requirement is of the order n, where n is the number of paths (preceding words) kept within a single word hypothesis. This n is much smaller than N, which is the total number of sentence hypothesis. [11] shows that the results achieved by this algorithm is same as exact N best with much lower computational overhead.

2.12 Multi Pass Decoding

Single pass decoding uses the viterbi beam search algorithm and computes the most probable sequence of phones given the observed speech. This sequence of phones usually corresponds to the most probable sequence of words. But, this assumption may not be true in case the probable path consists of a word in the dictionary with multiple pronunciations. Also, the viterbi algorithm assumes the dynamic programming invariant, i.e. if the ultimate best path for the observation passes through state K, then it must include the best path upto and including state K. Higher N-gram language models provide better accuracy by attaching probabilities for sequences of words. However, these models violate the dynamic invariant and we need to backtrack to find the best path or risk losing this information.

In Multi Pass Decoding[12], we break the decoding process into two passes. In our case, we implement the first pass in hardware using unigram/bigram models which attach probabilities to individual words only, and do not violate the dynamic programming invariant. This eliminates the need for backtracking and simplifies the hardware implementation. Also, it allows us to use a wide beam width and a larger vocabulary for the viterbi search. The output of this coarse first pass decoding step is a lattice of identified words. This lattice includes multiple paths and not just the best path. This lattice is of a much lower order, compared to the entire word vocabulary. The second decoding pass which is implemented in software uses this word lattice as the input. It applies language weights from more accurate N-gram language models to this lattice, in order to get the best hypothesis.

Using Multi-Pass decoding allows us to have a hardware implementation using a unigram/bigram model and very large word vocabulary. This implementation is very

generic and can be used as the coarse first pass for speech recognition in any application. We just need to make sure that the beam width for this pass is wide enough, so that it always includes the best possible hypothesis in the word lattice. The second pass of decode is in software and can be very application specific. For example, it can have an N-gram for, "Call XYZ from ABC on cell phone". The multi-pass decoding approach is very well suited for a hardware software co-design because it removes helps keep the first decode pass generic and does not require any feed back from the second decode pass. This allows the front end, senone scoring and both decode passes to work completely in parallel.

2.13 Fast Match

The N-best algorithm requires a fast match to quickly find words that are likely candidates for matching some portion of the acoustic input. This fast match is achieved using a lexical tree. The root of the tree is a context independent phone and the leaves are all the words which can start with that phone. We use this technique to activate new words in our hardware. We use a bit map of the fast match tree and call it the word activation map.

2.14 Word Lattice

A word lattice is a directed graph. We use this lattice as the output of first pass of decode. It represents the information about possible word sequences. The nodes of the graph are words. For each word, we store the previous word from which we transitioned to this word. This gives the path information for the lattice. In addition to this, we store

the accumulated score for the word. There can be more than one entries for a single word in the lattice if it lies on different paths. Thus, it is also necessary to store the start and end time for the words.

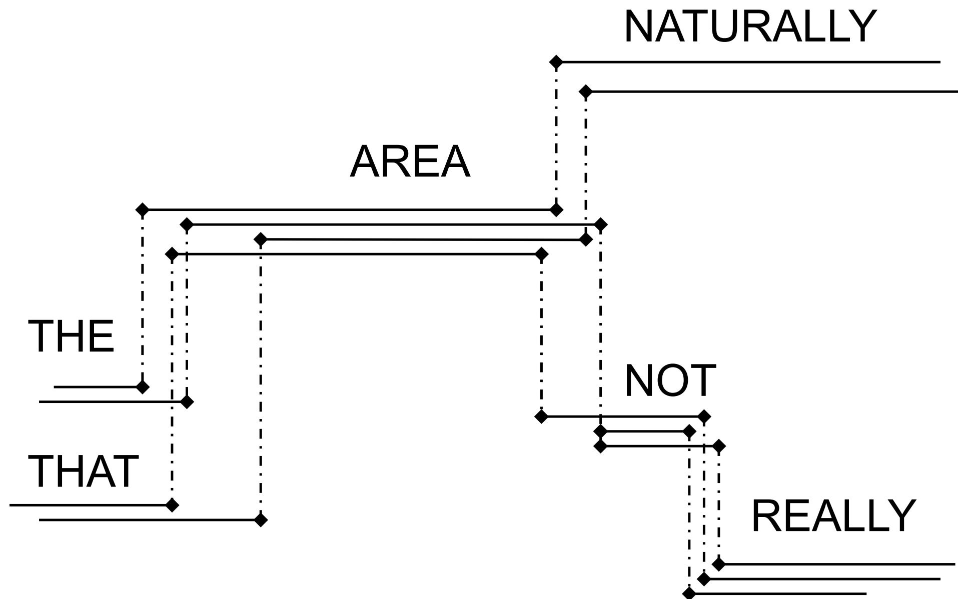


Figure 2.4: Example of a Word Lattice

2.15 Stack Decoding

A* or stack decoding is a one pass alternative to multi pass decoding [12]. It is a best first search of a tree that defines the sequence of allowable words in the language. It avoids the Viterbi approximation of choosing the best path. All the possible sentences in the language model are structured in the form of a lexical tree. This algorithm begins the search the start (root)node of the tree and moves towards the leaves, looking for the highest probability path. The paths in the tree are the language weights and the nodes

can be words or context dependent phones. The probability of each node is calculated using acoustic modelling. The A* algorithm finds the best path by keeping a priority queue of all partial paths calculated till any point in time. The algorithm then chooses to expand either the best path or the best N paths.

Chapter 3

Related Work

3.1 Existing Software Solutions

Many speech recognition solutions today are software implementations. CMU Sphinx [14] is a software solution developed by Carnegie Melon University. Microsoft Speech SDK and Nuance Dragon Speech are some of the commercial Software solutions available today. A Major problem with software solutions today is the excessive memory bandwidth and computing power required to obtain high recognition accuracy in real time. Use of smaller acoustic models and word dictionaries to maintain real time performance induces inaccuracy in recognition. The high computational requirement uses up most of the resources on a general purpose CPU and the acoustic models use most of the cache and DRAM. This results in resource contention and leaves the CPU unable to do any other task along with speech recognition.

The computation bottleneck for observation probability calculation can be solved by using increase parallelism in a hardware accelerator or by using a faster processor. The memory bottleneck for acoustic model read can be used by using a high band width

memory and by using algorithmic improvements. We use a combination of both. The memory bandwidth can also be reduced by reducing the data width (precision) of the acoustic model data. Many previous works discussed in the next section show that this does not affect recognition accuracy, while some of these works claim otherwise. Hence we choose not to use this optimization in our proposed hardware.

For the search algorithm, computation is a simple compare and add, and hence not a bottle neck. However, the random nature of accesses to HMM, word and language models poses a huge memory bottleneck. We solve this bottleneck by partitioning the sequential and random data and access each of them independently, to improve our memory access efficiency. This optimization provides us with majority of our performance and power improvements.

3.2 Existing Hardware Solutions

Existing Hardware solutions [6, 2, 20, 3, 21, 22, 5, 23] are far from a generic processor and are optimized for use with a given set of acoustic and N-gram language models. Many of them are unusable with other models while others suffer considerable performance degradation when used with other models.

Existing Hardware Software Co-Designs [24, 1, 25, 26] mainly calculate the observation probability in hardware. However, it is unclear as to which tasks in the algorithm should be fixed in hardware and which parts should be software controllable through commands, to allow maximum flexibility while providing improved performance.

Binu et al. [1] was one of the first to propose a hardware accelerator for speech recognition for Sphinx 3.0. They concentrate on accelerating the Observation Calculation Function. Their results show that using a reduced mantissa width of 12 bits does not

cause any reduction in accuracy. They obtain a huge performance improvement by using this reduced precision data format for arithmetic operations and acoustic models. This greatly reduces the memory bandwidth requirements as well. They claim a 104 times reduction in energy consumption over a $0.13\mu m$ Intel Pentium 4 Processor. They use energy delay product as a performance metric for comparison with other solutions. In our work, we do not concentrate on speed improvement, we just make sure that the recognition hardware performs real time.

Pazhayaveetil et al.[20] once again concentrates on hardware acceleration for observation probability calculation. They propose a scheme to skip senone scoring for frames to reduce the bandwidth requirement. This skipping is based on the distance between scores of consecutive frames. They also propose GMM level improvements like selective gaussian scoring, which skips gaussians which have generated scores below a certain threshold in previous frames. They also use codeword and clustering method to calculate senone scores using a reduced set of gaussian mixture values. They claim a 60% power reduction after using all these techniques, compared to their base design. However, they do not comment on any inaccuracies that may be caused by these techniques. In any case, the performance improvements obtained by these techniques should be orthogonal to the improvements proposed in this work, as we obtain our performance improvement by improving memory access efficiency and using a simple language model.

Cheng et al.[25] implements the GMM accelerator on FPGA uses software for the decode stage, this allows for use of larger acoustic models but limits the size of the language model that can be used. They use the adaptive beam pruning technique proposed in [13]. We use a very simplified version of this technique, since we found that a simplified first order approximation of the technique did not introduce and inaccuracies in out word error rate. They obtain performance improvement for GMM calculation by using high

parallelism in the data path and a high speed SDRAM.

Choi et al.[4] proposes a hardware solution for a 5000 word speech recognition task. They use a technique called sub vector clustering which shares gaussian mixture data into clusters to reduce the memory bandwidth requirement for Observation Probability Calculation. They evaluate multiple partitioning schemes for sub vector clustering. The performance improvement obtained by this technique is also orthogonal to the performance improvement obtained in this work. Also, they use 16 bit fixed point representation for the acoustic model, which further reduces memory bandwidth requirement. The improvement obtained by reducing data format width can be use along with the improvements proposed in this work.

Choi and You[5] proposes an FPGA based architecture for 20000 words. They us a DRAM and a BRAM in their architecture. They implement the sub vector clustering technique from [4]. They show that there is negligible loss in accuracy if 8 bit data formats are used for acoustic models. This results help them in greatly reducing the memory bandwidth requirement. They calculate scores for active gaussian mixtures from adjacent frames together, in order to reduce the memory bandwidth requirement. We use a similar strategy for two adjacent frames. However, we compute the scores for a block of two frames. In our case, we score the entire acoustic model over this block of two frames. This not only reduces the average memory bandwidth, but also the peak memory bandwidth, as there is no change in the number of gaussian mixtures computed for each frame. [5] also propose an optimized DRAM access scheme for inter word transitions in the language model and take advantage of variable length burst BRAM accesses. They introduce a scheme to fetch small parts of the word model with variable length bursts. This allows them to fetch only the active HMMs of the word and reduce memory bandwidth. In our case, we store the HMM models and word models separately. The

word models consist only of HMM IDs, which are later used to fetch the HMM data only if the HMM is active. Thus, our data partitioning and storage scheme achieves similar performance improvement.

Chandra et al.[6, 2] proposes an ASIC for observation probability calculation and HMM calculation. The word decoding is performed in software. This solution provides real time computation capabilities but requires very high communication bandwidth from the hardware to CPU since the scores of all the HMMs have to be sent back to CPU. Also, this approach does not optimize the high memory bandwidth requirement for the activation of new words in the the word search. The word decode stage in software generates a word lattice which is similar to the lattice generated in this work. The language model is then used by another software stage which picks the best path from the word lattice.

Li et al.[26] proposes a low power architecture the observation probability calculation using FPGA. This solution relies on parallelism and multiple SRAMs for increased performance. They concentrate on dictionaries with less than 1000 words and use a CHMM based acoustic model. We try to achieve a similar performance benefit using a high bandwidth NOR flash memory. We will also see in chapter 6 that the power consumption of our proposed system is lower.

Bourke et al.[27] proposes a hardware solution for the search section of the algorithm. Our architecture uses a similar architecture but implements the multipass decoding algorithm which provides increased flexibility. Also, we use an approximate word dependent N-best search [11] to obtain better accuracy, without the computational complexity of a full blown exact N-best search [10].

Bourke et al.[23] uses an architecture similar to [27] but works on a smaller word vocabulary with a reduced beam width. This allows them to store the entire active list

on a 11.7Mb on chip SRAM, which improves their performance and greatly reduces power consumption.

3.3 This Work

In this work, we propose a scalable and portable hardware accelerator for speech recognition. It accelerates the acoustic modeling and decoding process of the speech recognition algorithm. We use the multi-pass decoding approach [12], which splits the decode process into two parts. The first pass is carried out on a large search space, using a simple language model and an N-best Viterbi Search [10, 11] , which works reasonably well at coarse recognition and reduces the size of the search space. The second pass is carried out on a smaller search space, using more sophisticated N-gram [8] language models. We implement the first pass in hardware and the second pass in software. This keeps the performance of the hardware unaffected by the use of sophisticated models without compromising on end accuracy of the recognition process. Moreover, it makes hardware design much simpler. We choose the Sphinx 3.0 [14] speech recognition software developed at Carnegie Melon University, to demonstrate the benefits of our proposed architecture.

Chapter 4

Proposed Hardware Architecture

In this chapter, we will discuss our proposed architecture for hardware portions of the algorithm. We start with design space exploration which helped us choose our optimal design point and then we propose the hardware architecture to meet this accuracy requirement in real time.

4.1 Design Space Exploration

We use the SystemC model of the hardware with CMU Sphinx 3.0 [14] for design space exploration and performance analysis of the speech recognition system. The use of a parameterized SystemC Model allows us to investigate the trade offs for various HW/SW partitioning schemes. It also allows us to observe the impact of various hardware configurations on the overall performance of the system. We model both the hardware and the communication interface in order to observe the end-to-end latency for any operation.

There are multiple hardware software partitions possible for an HMM based speech recognition system. We can choose to accelerate the observation probability calculation

stage [6, 20, 25] in hardware while the word search is performed in software(*split*1). The next step is to accelerate the processing of HMMs (triphones). In this split (*split*2), the HMM scores are sent back to the software every frame. The software performs the transitions from one HMM to the next. In *split*3, the hardware performs transitions for HMMs within a single word, and the inter word transitions are performed by software. The hardware sends back a list of identified words to the software and receives a new list of words to be activated every frame. The next step is to bring in the inter word transitions into hardware (*split*4). Table 4.2 shows the various HW/SW splits we investigated before we started our hardware design.

Table 4.1: Hardware Software Partitions Investigated

Partitioning Scheme	**All Software**	**Split 1**	**Split 2**	**Split 3**	**Split 4**
Senone Scoring	SW	HW	HW	HW	HW
HMM Scoring	SW	SW	HW	HW	HW
Word Scoring	SW	SW	SW	HW	HW
Word Transitions	SW	SW	SW	SW	HW
Second Pass of Decoding	SW	SW	SW	SW	SW

For performance comparison, we assumed that every task which is off loaded to Hardware can be performed in real time and the HW and SW can run completely in parallel. The software used was sphinx 3.0 running on a 2.4 GHz Intel Core 2 Duo processor. Software performance was calculated using built in performance counters in Sphinx. The HW/SW interface performance was estimated using the interface model discussed in section 5.12. All hardware performance, data access patterns and bandwidth requirements were calculated using the SystemC model discussed later in chapter 5. We used an acoustic model for 8000 senones, 16 gaussians [28] and a tri-gram language model with

64K words [29] for this exercise. The goal was to observe the performance improvement and communication bandwidth requirements for all possible HW/SW partitions, while keeping the word error rate constant. Fig. 4.1 and table 4.2 show the results of our simulation. It can be observed that as we bring more and more tasks into hardware, we achieve better performance improvement. We can see that off loading of inter word transitions ($Split4$) into hardware provides us with a huge performance benefit, since it involves lot of memory accesses. The improved performance translates to better end accuracy of speech recognition by allowing us to use larger acoustic and language models in real time.

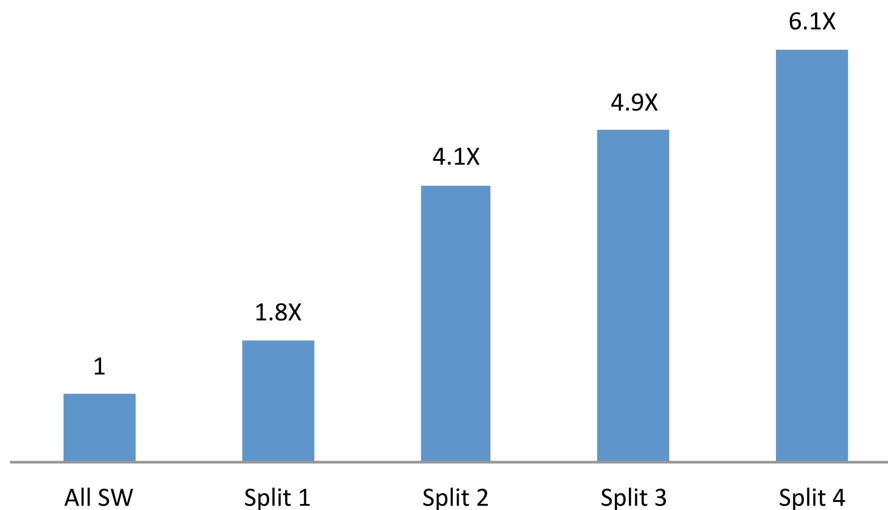


Figure 4.1: Performance comparison with sphinx 3.0 running on desktop PC with various HW/SW splits for acceleration

Another factor which is important in a HW/SW co-design is the communication bandwidth requirement between the hardware accelerator and CPU. In our case, this parameter has even more importance as our system sits on a peripheral bus. Table 4.2

shows the communication requirements for various HW/SW splits. *Split*1 which is implemented in many systems today [1, 6, 20, 25] requires nominal communication bandwidth. This system provides good performance improvement for large acoustic models but not for large language models, as most of the time is spent by software in doing the word search. *Split*2 requires large communication bandwidth and requires a very fast interface. *Split*3 is very efficient in terms of bandwidth requirements, however, inter-word transitions which are memory intensive and can benefit from hardware acceleration, are still done in software. Moreover, in this partitioning scheme, to maintain sequential DRAM access in hardware (this provides huge performance boost), the incoming list of words to be activated has to be sorted by software to match the order of the active word list in hardware. This results in additional software overhead. *Split*4 offers both low communication bandwidth and high performance improvement. Moreover, this scheme allows the hardware to run in parallel to the software, without any feedback from software. The output of the hardware is a reduced search space (word lattice) on which the software can work in the second decode pass, to find the best hypothesis. We chose to use *split*4 for our hardware design.

Table 4.2: Communication Bandwidth Requirement for various HW/SW Splits

Partitioning Scheme	**Communication Bandwidth**
Split 1	3.22 MBps
Split 2	84 MBps
Split 3	0.76 MBps
Split 4	0.156 MBps

For a high accuracy real time speech recognition it is necessary to support large acoustic models and large word vocabularies. Another factor which affects accuracy

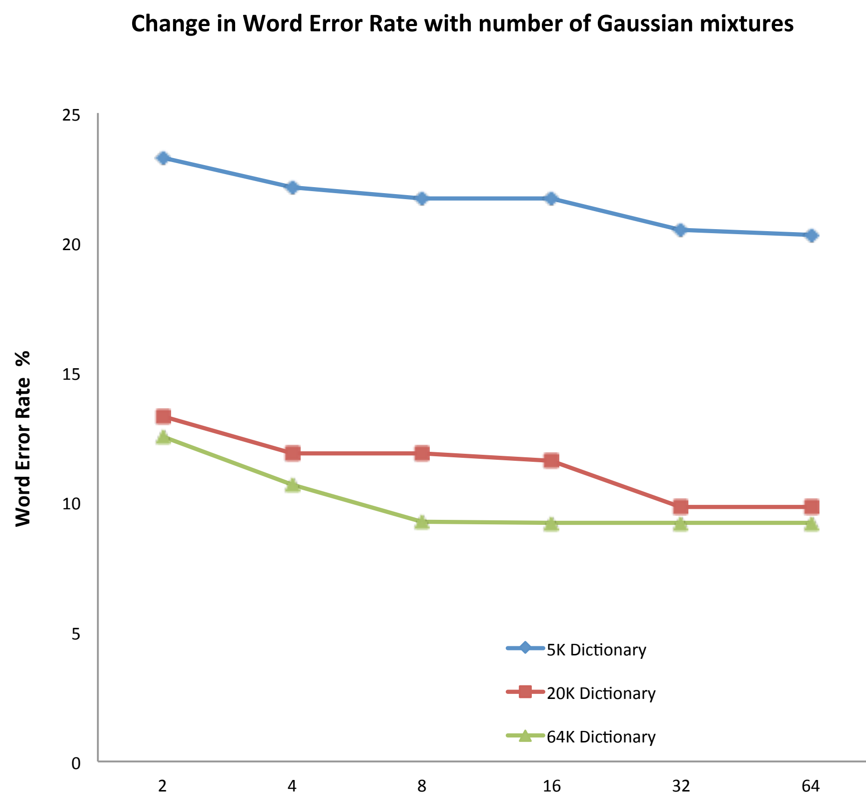


Figure 4.2: Effect of Number of Gaussian Mixtures in the Acoustic Model on Word Error Rate

is the use of sophisticated language models which model probabilities of sequences of words, rather than individual words. As seen in Fig. 4.3, a trigram language model performs better than a bigram model. It is difficult to build hardware which is optimized to work equally well with any N-gram language model. Working with larger N-gram models increases number of random accesses to memory. It also greatly complicates the architecture as we now need to keep track of history for last N words and back track for the best path if necessary. In addition, we also need searching or hashing techniques to determine if an N-gram is present in the model for a given active word sequence.

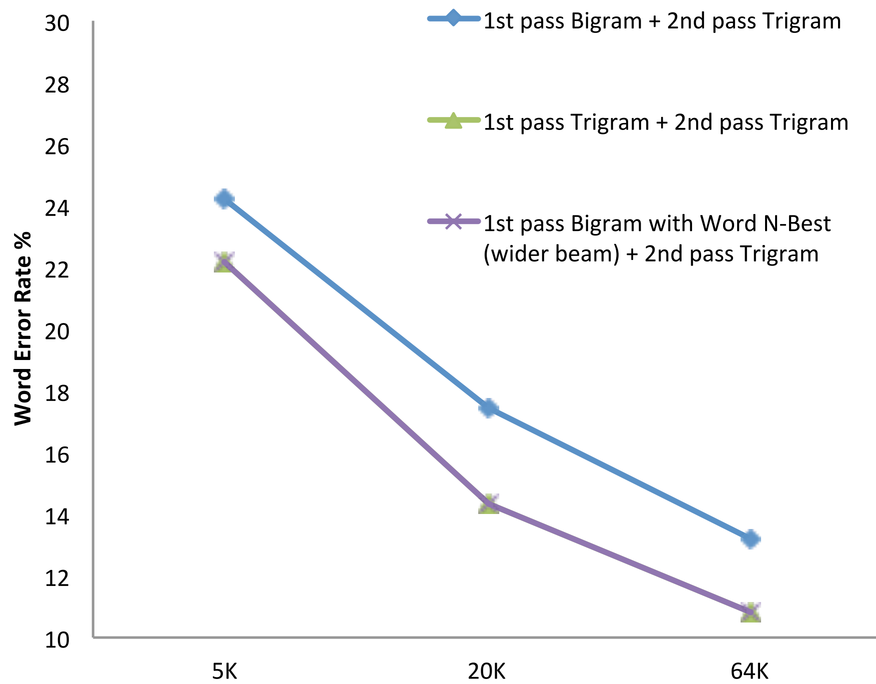
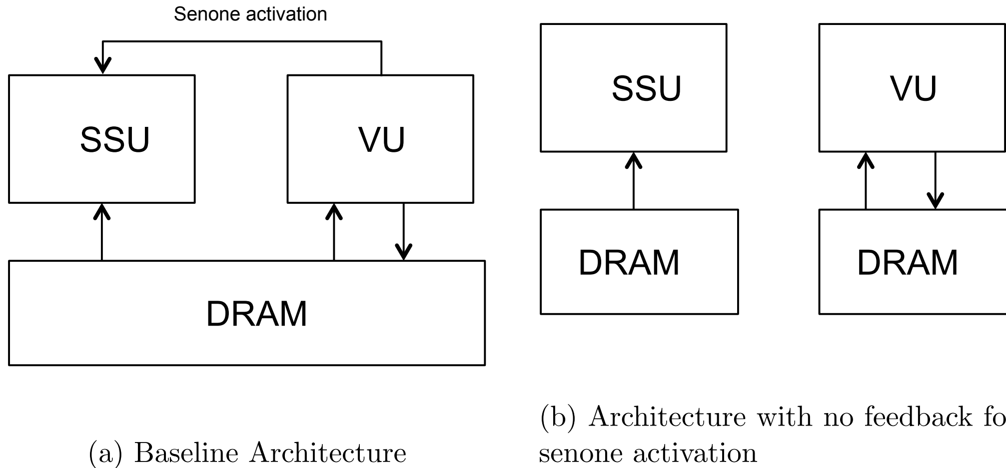


Figure 4.3: Word Error Rate for different language models in Pass I and Trigram in Pass II



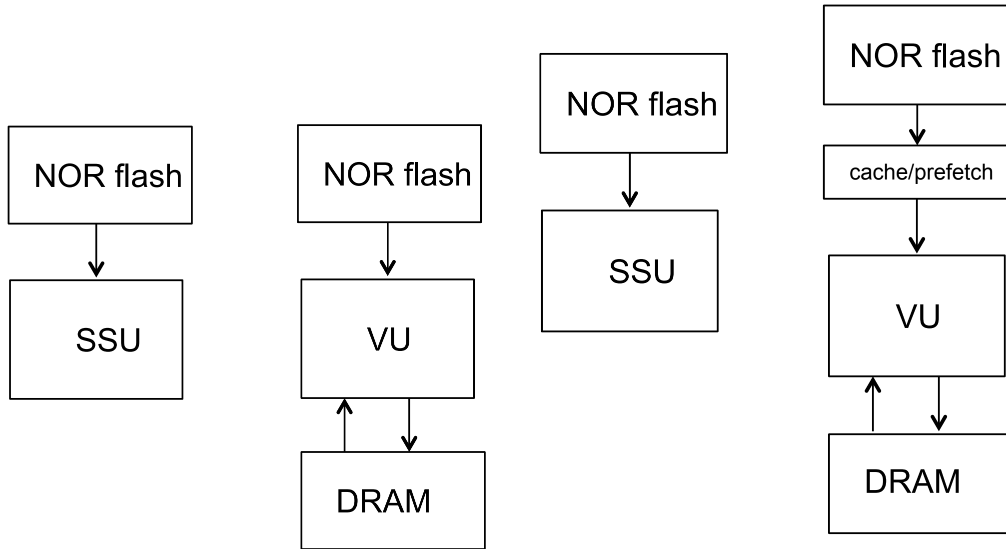
(a) Baseline Architecture

(b) Architecture with no feedback for senone activation

Figure 4.4: Hardware architechtures explored I

We choose to design our system to work with unigram/bigram language models, allowing us to reduce hardware complexity. Unigrams can be indexed directly using word IDs and Bigrams can be indexed using either the source or destination word ID. Direct indexing keeps memory accesses very simple. We implement the multipass decode approach [12] where the first decode pass is a word dependent approximate N-best time synchronous viterbi beam search algorithm [10, 11] which returns a word lattice representing multiple best paths rather than a single path. We increase the beam width for the first pass which uses an bigram model, making sure that the best hypothesis is not omitted from the output of the first pass. The second pass using trigram language model works on the word lattice generated in the first N-best pass and chooses the best hypothesis.

We started with a baseline design for the hardware as shown in Fig. 4.4a where the senone scoring and decoding share the same memory and do not work at the same time. The decoding stage provides feedback to the senone scoring stage, which activates senones which need to be scored for the next frame. As seen in Fig.4.7, for a system



(a) Architecture with NOR Flash for acoustic and language model storage

(b) Architecture with Cache for HMMs and Prefetch for Bigrams

Figure 4.5: Hardware architechtures explored II

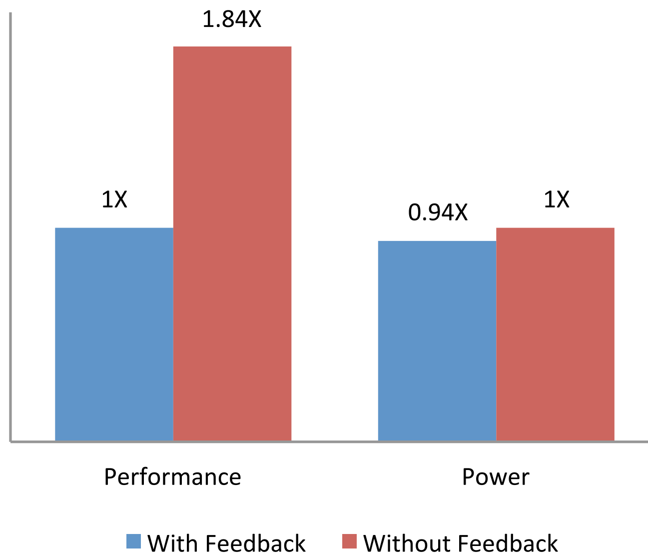


Figure 4.6: Comparison between hardware accelerator with and without feedback for senone activation

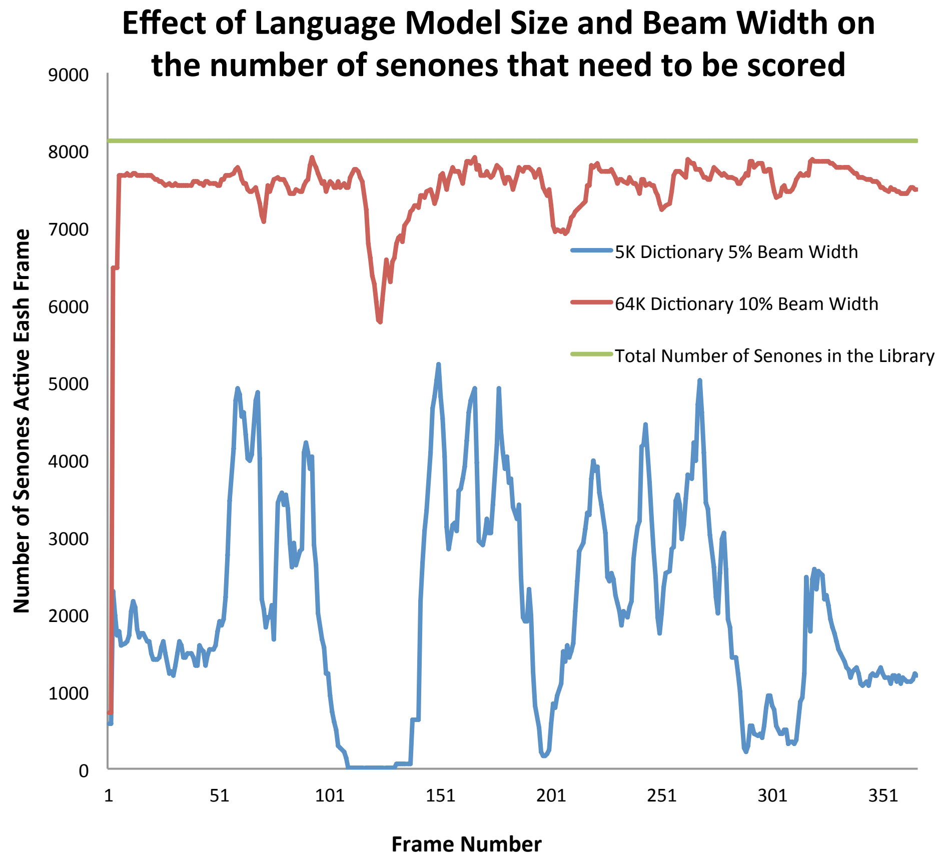


Figure 4.7: Number of Senones Active for each frame for 5K dictionary with narrow beam width and 64K dictionary with wide beam width

which uses a wide beam width (larger than 10%) and a large vocabulary of (64K words), 92.2% of the senones are always active, as compared to 25% for a smaller vocabulary of 5K with a narrow beam width. The activation of senones for each frame not only introduces added complexity and memory requirement, but also introduces dependency between the decode stage and the acoustic modelling stage. We break this feedback loop and decide to score all the senones for every frame. Although we calculate 8% more senones, it allows the senone scoring to be performed independently and in parallel to the decode stage. While the decode stage works on frame N, the senone scores for next frames can be calculated. This architecture is shown in Fig. 4.4b. Breaking this feedback offers significant performance improvement for a modest increase in power consumption (see Fig. 4.6).

Table 4.3: Data accessed every frame for senone scoring and decode pass I

	Volatile	Non-Volatile
Senone Scoring (8000-senones, 8-mixtures, 39-features)	156 B	24.6 MB
Viterbi Decoding (64K-Bigram)	1.42 MB	3.18 MB

For the the senone scoring stage, the entire acoustic model has to be read for each frame. For the decode stage, we need to read the HMM structure data and language model weights for each active HMM. In table 4.3 we can see that 86% of the data accesses are non-volatile. Hence, we opt for the logic on memory approach with an on chip high bandwidth NOR flash memory (Fig. 4.5a). We use a combination of on-chip SRAM and off-chip DRAM for storing the non-volatile data like senone scores and active HMM scores. We performed simulations to determine the access efficiency of multiple

memory configurations (see Table 4.4). For Senone Scoring, since we read the entire acoustic model sequentially, the DRAM provides good access efficiency. The Flash still performs better because it does not need activation, refresh and pre-charge. For the Viterbi decoder, the access to word models, HMM models and Bigrams are random. Hence, we greatly improve our efficiency by storing these models in a high bandwidth NOR Flash Memory.

Table 4.4: Memory Access Efficiency for different Hardware configurations

Configuration	Efficiency
Senone Scoring with DRAM	78%
Senone Scoring with Flash	94%
Viterbi Decoding with DRAM	48%
Viterbi Decoding with Flash + DRAM	89%

Two factors make us move to the design in Fig. 4.5b. For the decode phase, we read the word model / HMM model only once per word and cache it, so that it can be reused for the remaining active instances of the word. Fig. 4.8 shows us that with increase the number of instances for a word, the cache hit rate increases dramatically. The optimal size of the cache was found to be as small as five HMMs within a word. We purge the entire cache from one word to the next. Higher cache hit rate means that the available flash bandwidth can be used to prefetch multiple rows of bigrams for word transitions. This masks the bigram fetch latency and improves performance. This is the architecture we implemented and will be discussed in upcoming sections.

We performed simulations on 400 utterances for the Wall Street Journal CSRI corpus, using the proposed multipass N best decode approach. The results of these simulations are shown in Fig. 4.2. From Fig. 4.2, we can see that a large 64K vocabulary and

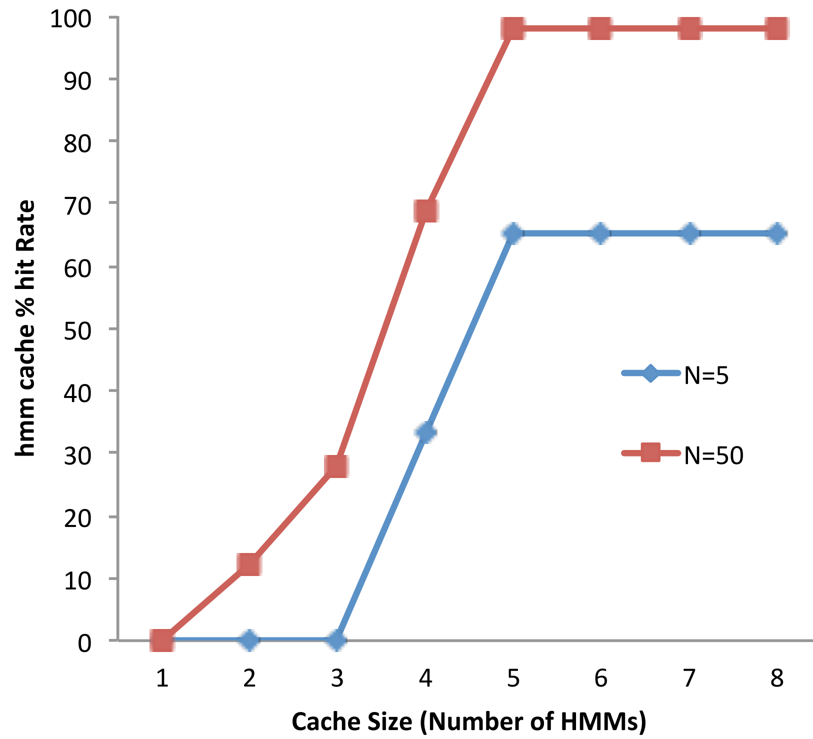


Figure 4.8: HMM cache Hit rate for various cache sizes and values of N (number of hypothesis kept active at every word)

an acoustic model with 8 gaussian mixtures is the optimal design point. We want to support a large vocabulary and 64K was the largest set of trained language models we found in the open source community. We see that increasing the number of gaussian mixtures beyond 8 did not provide much improvement in word error rate for the 64K vocabulary. Hence we chose 8000 senones, 8 gaussians and 64K word Bigram as our design point. We designed our hardware to meet this requirement in real time, including the communication between hardware and CPU.

4.2 Overall System Architecture

In this section, we will describe the basic blocks of our proposed hardware accelerator. The system consists of three main units (see Fig. 4.9). The Interface Control Unit decodes the commands and data obtained from software and controls the functioning of the Senone Score Unit (SSU) and the Viterbi Unit (VU). It is also responsible for sending the word lattice back to the software when it requests for it. The SSU calculates the observation probabilities, i.e. senone scores, for the entire library of senones. The VU calculates the state probabilities i.e. state scores, for all active states in the language model. In addition to this, it applies pruning thresholds to the states and activates new states if needed. It also adds recognized words to the word lattice as and when they are recognized.

4.3 Hardware Software Partition

As shown in Fig. 1.2, we choose to perform the front end DSP and second stage of decode in software. Both are implemented using sphinx 3.0 [14]. The front end from Sphinx that

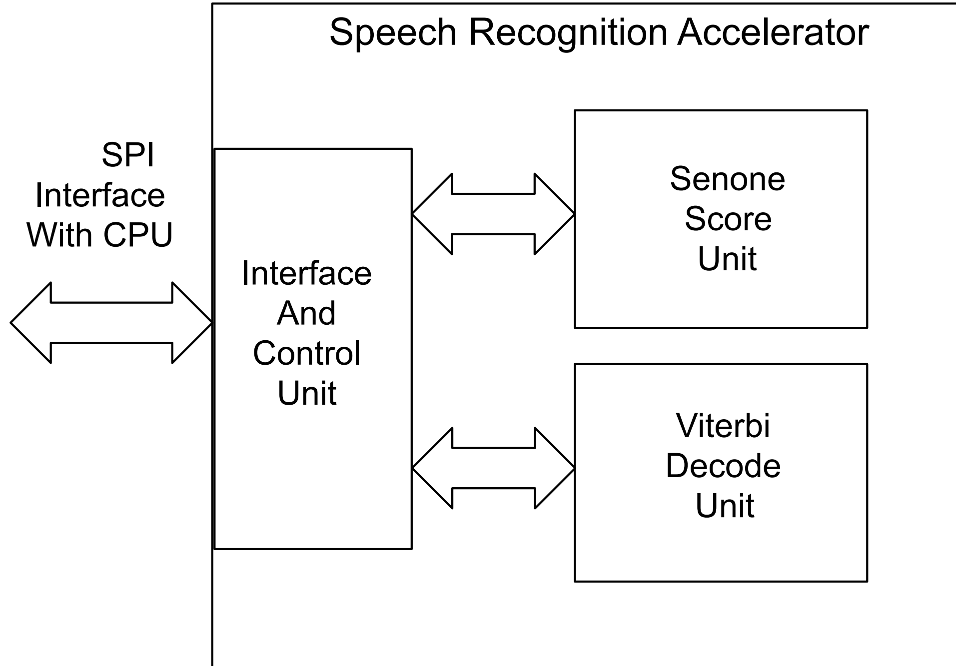


Figure 4.9: Top Level Hardware Block Diagram

we use is a 39 dimensional feature vector with MFCC coefficients. The second pass of decode is performed by running sphinx in the word lattice input mode. Senone Scoring and First decode pass using N-best search is performed in hardware as proposed. The input being the feature vector generated by sphinx front end and output being the word lattice, which is passed on to Sphinx again, for the second decode pass.

4.4 Software Commands

Our choice of interface between CPU and hardware is a 50MHz dual serial peripheral (SPI) bus, as this is increasingly becoming the interface of choice for NOR Flash memories. The interface and control unit decodes the commands received over this interface. The SSU and VU are used to service these commands. We define the following set of

commands for our hardware.

SET_ACOUSTIC_MODEL

This command is accompanied by the memory offset for the acoustic model we want to use. This lets us change acoustic models on the fly.

SET_LANGUAGE_MODEL

This command is accompanied by the memory offset for the language model we want to use. This lets us switch language models. For example, we can have separate language models for dictation and command control.

SET_HMM_INIT_BEAM

This is a score based threshold used to define the widest beam width to be used for the Viterbi search.

SET_WORD_INIT_BEAM

This is a score based threshold used to define the widest beam width to be used for word exits.

SET_MAXHMMPF

This is a very important parameter for any hardware implementation for speech recognition. This sets the maximum number of HMMs we want to keep active for any frame. This defines the memory requirement and performance of our system. We use a technique called adaptive pruning [13] to calculate the threshold using the number of active HMMs

from previous frames. The threshold is adjusted to keep the number of active HMMs close to *maxhmmpf*.

SET_MAXWPF

This parameter limits the maximum number words we can add to the word lattice in a frame. Reducing the value for this parameter implies sacrificing accuracy as this governs the number of average number of best paths we maintain from each word, for the first pass of decode. We use the adaptive pruning [13] technique to calculate the word exit threshold which keeps the number of words recognized each frame close to *maxwpf*.

SET_MAX_N_BEST

This parameter sets the maximum instances of a single word we want to follow. Since we need to keep all the entries in a buffer till all the path for a word are processed, the DRAM read buffer size directly effects the maximum value supported here. In our simulations we set this value to 10, however, we discovered that the average number of paths that were above threshold was 2.9 for ensuring that the best hypothesis was withing the word lattice.

SET_FEATURE_LENGTH

This parameter is used to dynamically change the dimensionality of the incoming feature vector from software. Thus, we can be compatible with multiple front ends [30]. This value has to match the feature length in the acoustic model. It is observed in [30, 31] that for lower input sampling rates, we can get away with using a lower dimensional feature vector. This helps us reduce our performance requirement without sacrificing accuracy.

SET_HMM_LENGTH

Using this command, we can set the number of states for each HMM in the language model. This is used for address translation in language model accesses in the Viterbi Unit.

SET_MAX_MIXTURES

Using this command, we can set the maximum number of gaussian mixtures used for any senone. This is used for address translation in acoustic model accesses in the Senone Score Unit. In addition to this, we can can also have a variable number of mixtures for each senone.

LOAD_FEATURE_BLOCK

This command is accompanied by the feature vector coefficients for a block of two frames. The SET_FEATURE_LENGTH command tells the hardware how many bits to expect with this command.

READ_LATTICE

This command is used to read the word lattice generated by the hardware after first pass of decode. The command is accompanied by the number of lattice entries the software wants to read. The hardware returns data with first byte as length of the list, followed by the word entries.

SET_UTTERANCE_ID

This command is accompanied by a unique utterance ID which tells the hardware that a new utterance has begun. It is needed so that an internal reset can be sent to the Senone

Score Unit and Viterbi Decode Unit.

INIT

This initializes the hardware. Default configurations and the log-add table are loaded from the Flash on to SRAM and registers. We need $650us$ to initialize. This operation needs to be done only once at start up.

PAUSE/RESUME

The software can use this command to instantaneously pause and resume hardware operation. No information is lost.

4.5 Hardware Output

The output of the first decode pass is a word lattice. The format of the lattice is shown in Fig. 4.10. As discussed before, each node in the lattice is a an identified word. This word is accompanied by the ID of the word it transitioned from. This is the path information. The acoustic score is used in the second phase for rescoring the paths in the word lattice. The start frame and the beginning and end frame of the last HMM of the word are used by sphinx to detect possible transitions from one word to another. These transitions are a soft boundary and the best acoustic score for a word could have potentially occurred anywhere between the start frame of the last HMM and end frame of the last HMM.

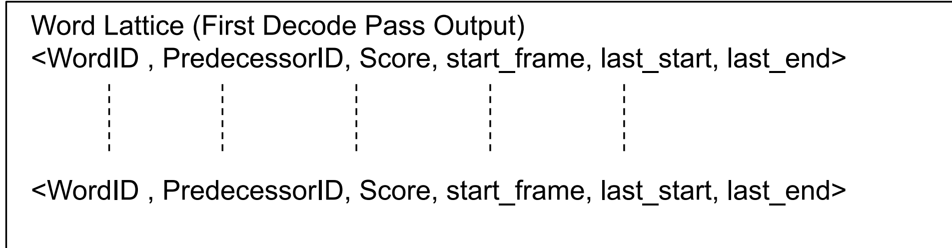


Figure 4.10: Word Lattice Output of First Decode Pass (Hardware)

4.6 Senone Score Unit

The Senone Score Unit (SSU) is shown in Fig. 4.11. It calculates the scores for each senone using equation 2.4. Once the start command is received, the senone ID incrementer loops through the entire list of senones in the library. This module is highly pipelined to provide sustained throughput. We also use multiple distance calculation units in parallel to consume all the data provided by the high bandwidth NOR Flash.

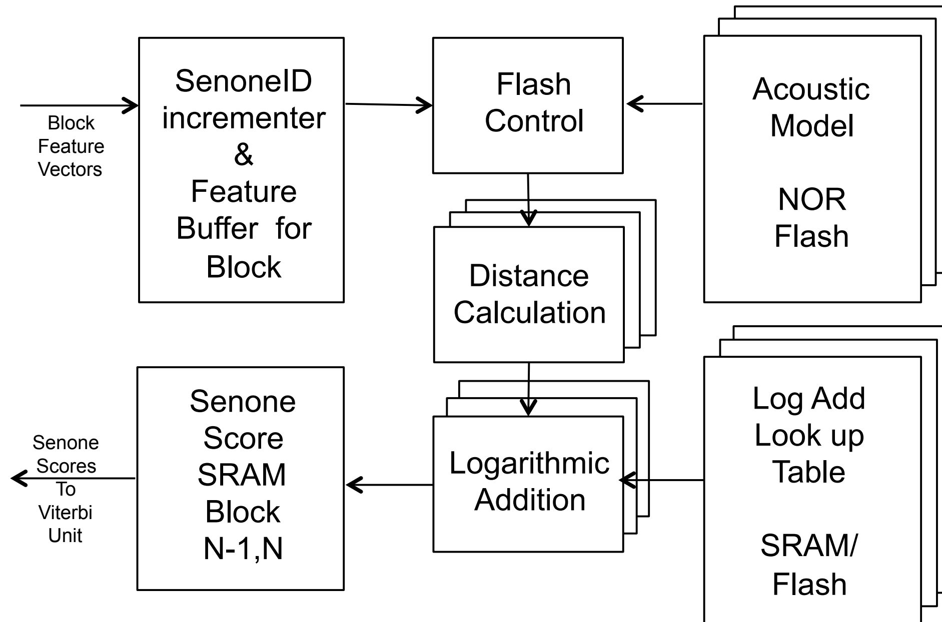


Figure 4.11: Block Diagram for a Scalable Senone Score Unit

4.6.1 Block Senone Scoring

The CPU sends a block of feature vectors for two consecutive frames to the hardware. The SSU computes the senone scores for both the frames simultaneously while reading the acoustic model just once. The calculation of scores is split across the entire length of two frames and hence does not require increased parallelism. The memory bandwidth for the NOR Flash is reduced by a factor of two. The size of the senone score SRAM increases by a factor of 2, since we now have to store the scores for two consecutive blocks, since the decode stage is still using the scores from the previous block as shown in Fig. 4.12.

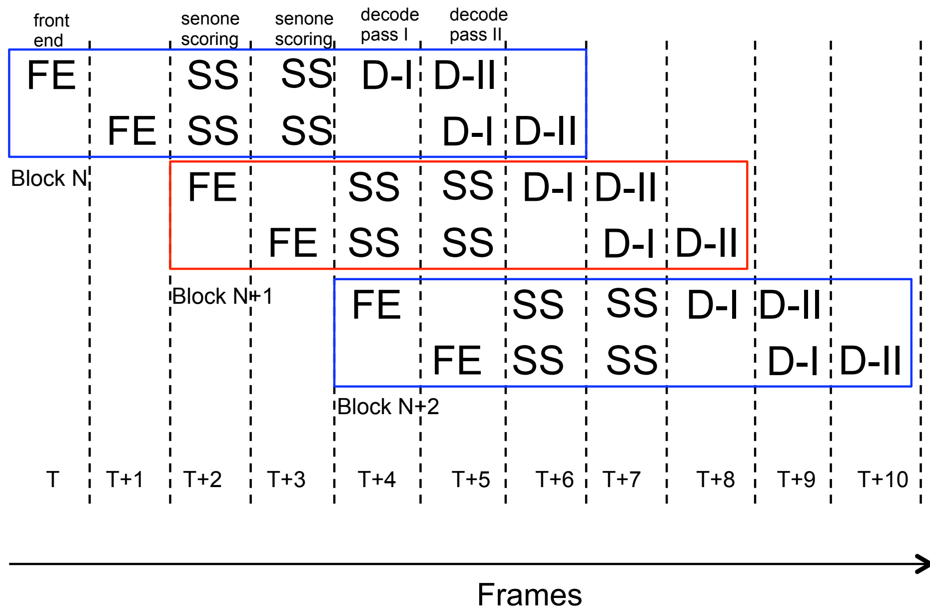


Figure 4.12: Pipelining of operations of the Speech Recognition Algorithm

Scoring of all senones for each frame implies that the feature vector is the only changing component every frame. This can be used to share acoustic model reads across

frames. This technique is a variation of sub vector clustering used in [4]. We spread the scoring of the entire senone library across multiple frames. Fig. 4.13 shows the affect of block size on the power consumption of the SSU. We use a block size of 2 since at our technology node, we are limited by SRAM size. Power estimations were obtained using the methods described in section 5.11.

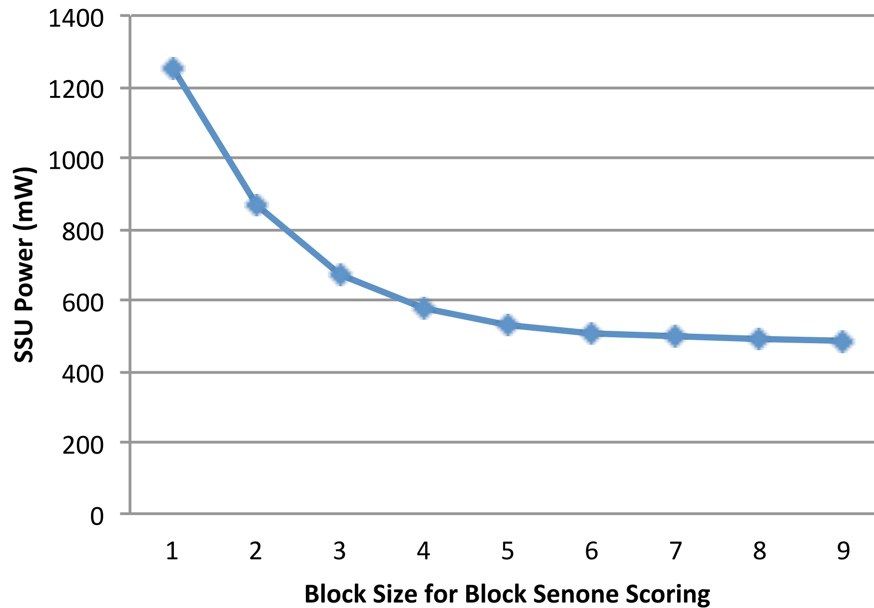


Figure 4.13: SSU Power Consumption for various block sizes for block senone scoring

We use a pipelined structure shown in Fig. 4.12 . We delay the decode stage by two frames. The senone scoring stage works on 2 sets of feature vectors simultaneously over the entire two frames while reading the acoustic model only once. This reduces our total memory reads, peak bandwidth requirement and subsequently the read power consumption. It adds a latency of two frames but still has real time throughput. The size of the senone score SRAM is increased by a factor of 2 since scores for the previously

scored block of frames are still being used.

4.6.2 Flash Control and Memory Structure

The Flash Control unit translates the acoustic library offset and senone ID into the first memory address of the senone entry. We use a packed data structure for senones in the library as shown in Fig. 4.14 , to ensure that the performance is limited only by the physical bandwidth of the memory. At the beginning of every senone, the length of each senone record is stored. This helps us identify the end of a senone. At the beginning of each library, we store the number of senones, which helps identify when we have calculated all senones in the library.

4.6.3 Distance Calculation

This module computes the inner summation for equation 2.4. It has 4 parallel units for the subtraction-square-multiply operations. The output of these units is used by 2 stages of addition.

4.6.4 Logarithmic Addition

Since, all the operations are in logarithmic domain, we need to do a logarithmic addition for the outer summation of equation 2.4. This involves calculating the value of $log(A+B)$ from $log(A)$ and $log(B)$. This is done using a look-up table similar to the one used by CMU Sphinx [14]. This unit needs pipelining as it has to access the lookup table. However, no parallelism is required since only one such operation takes place for N distance calculations.

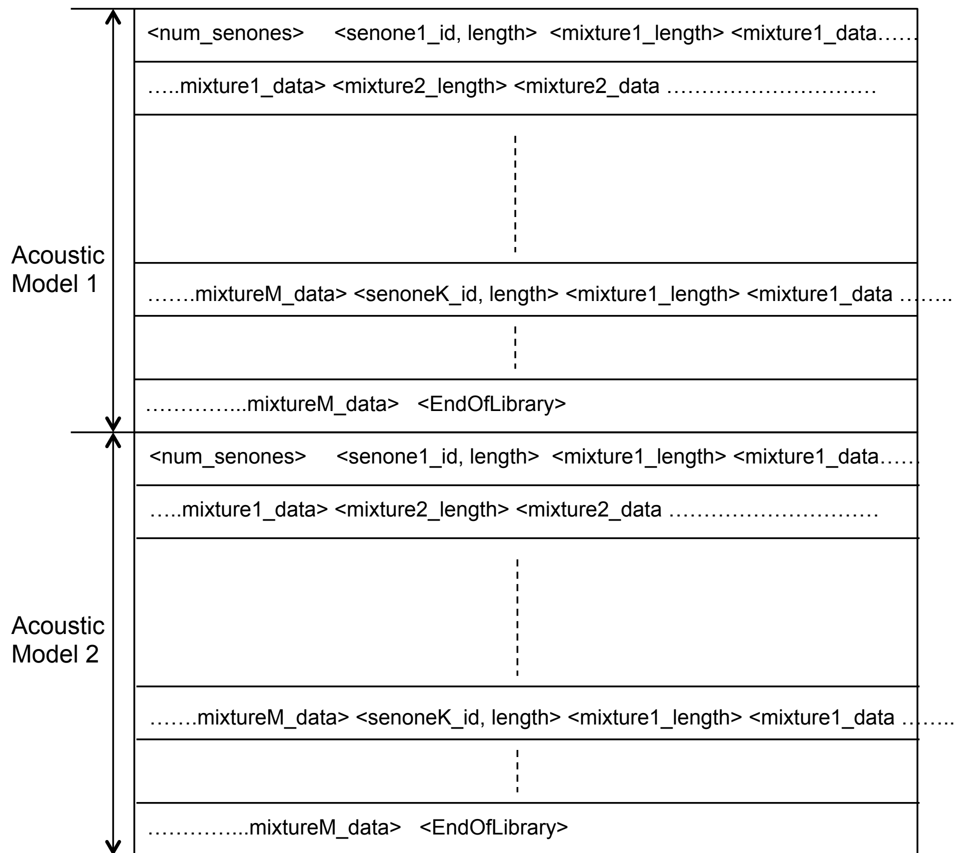


Figure 4.14: Packed Data Stucture for Acoustic Models in NOR Flash

4.7 Viterbi Decode Unit

The Viterbi Unit is responsible for performing the first decode pass on the incoming speech using a simple unigram/bigram language model and word dependent N-best search. It has a pipeline of 22 stages which works on each HMM from the active list, which is streamed from the DRAM sequentially. The random accesses required for the state score calculation in equation 2.7 are the transition probabilities $loga_{ij}$ and senone scores $logb_j(Y_t)$. We divide these accesses between the language model stored in flash and the senone scores stored in the SRAM by the SSU. The pipelining of these accesses provides us with a huge performance benefit.

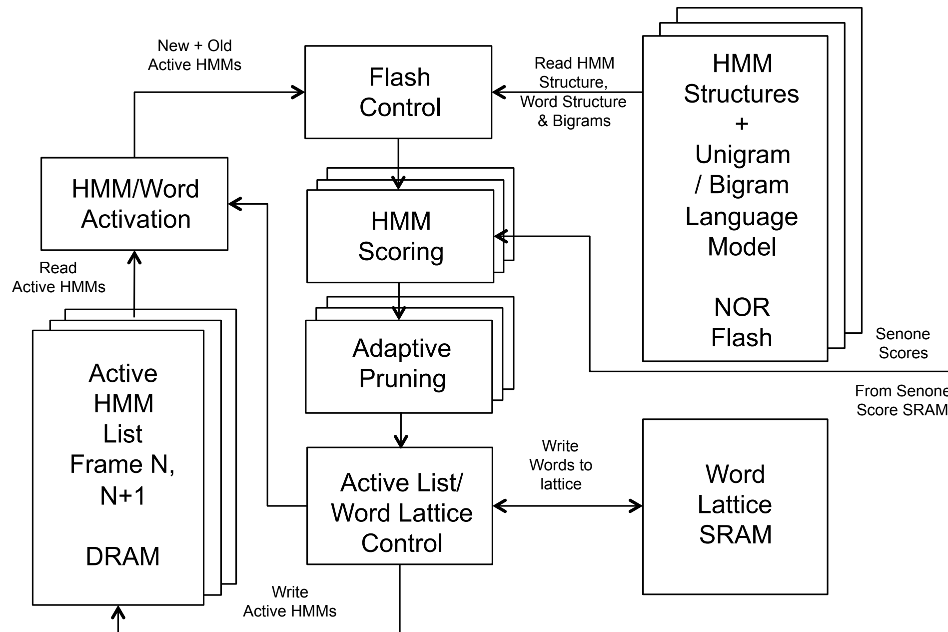


Figure 4.15: Block Diagram for a Scalable Viterbi Unit

4.7.1 Flash Control and Memory

We store the bigram language model, word structures and probabilities and the HMM structures in the Flash Memory. Each word occupies a line in a 256 bit wide line of memory and can be accessed in a single read. Similarly, an HMM upto 3 states (a triphone) can be accommodated in a line of memory and accessed in a single read. The word model and triphone structures are accessed only once per word and stored in a cache. The structure of our active list (section 4.7.4) allows us to reuse this data for multiple instances of the same word. The Bigrams are indexed by source word ID. We prefetch multiple lines of bigrams for each source word into the SRAM. The prefetch SRAM is divided equally to store multiple bigrams of all possible source words in that frame. The bigrams are read from this SRAM in the order of destination word IDs as each word from the active list is being processed.

4.7.2 HMM scoring

This block calculates equation 2.7 for each HMM. The last state scores are available from the HMM active list entry in DRAM. Transition probabilities and senones IDs are obtained from the flash. The senone IDs are used to read the corresponding senone scores from the SSU SRAM. We use a simple add-compare-select unit which adds the last state scores, transition probabilities and senone scores, compares them and then selects the best.

4.7.3 Adaptive Pruning

We use adaptive pruning [13] to limit the size of our search space for the first pass of decode. We start with the pruning threshold (T_0) set by the software and use equation

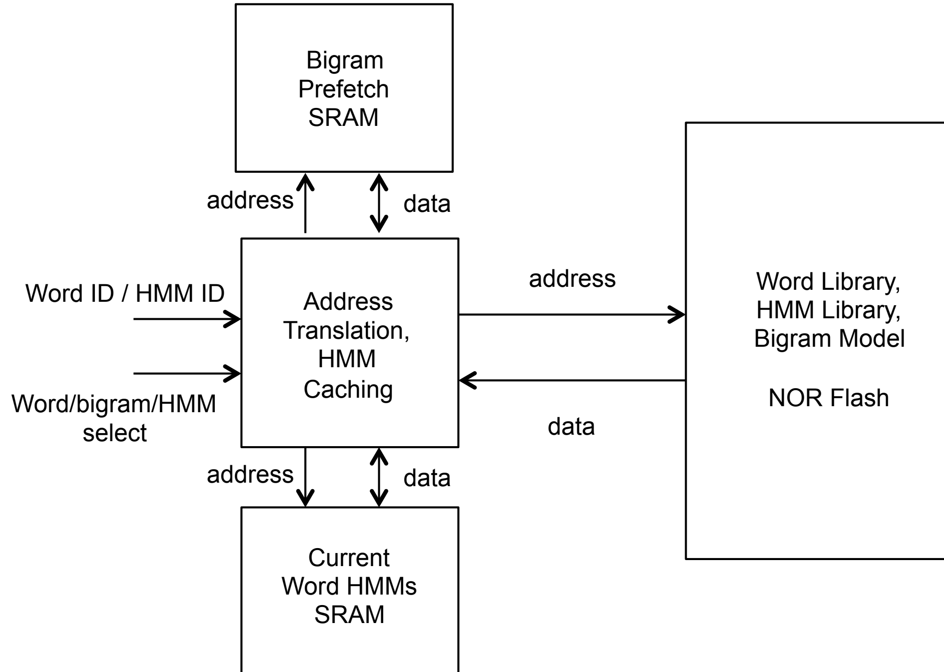


Figure 4.16: Flash Read Control for the Viterbi Unit

4.1 to calculate this threshold for each frame. Here, N_{set} is the $maxhmmpf$ parameter set by the user. N_t is the number of states that are active in the current frame. This equation represents a closed loop system, which adjusts T_{t+1} in order to keep N_{t+1} as close to N_{set} as possible. The value of α is set to 0.2 to dampen the response of this system. We add a 10% tolerance to N_{set} to make sure we do not pass less than N_{set} HMMs. This tolerance value was obtained empirically, by running 400 sentences from the Wall Street Journal CSR I Corpus. For any frame, we never use the adaptive threshold if it is wider than the initial beam threshold set by software. Also, we do not calculate the adaptive threshold unless the value of N_t is larger than N_{set}. A similar equation is used for the word thresholds and pruning of N-best paths as well. We subtract this threshold from the best HMM score for the previous frame t and compare it to every state score in $t + 1$ to prune them. For the word threshold, we do the same with the word score. For

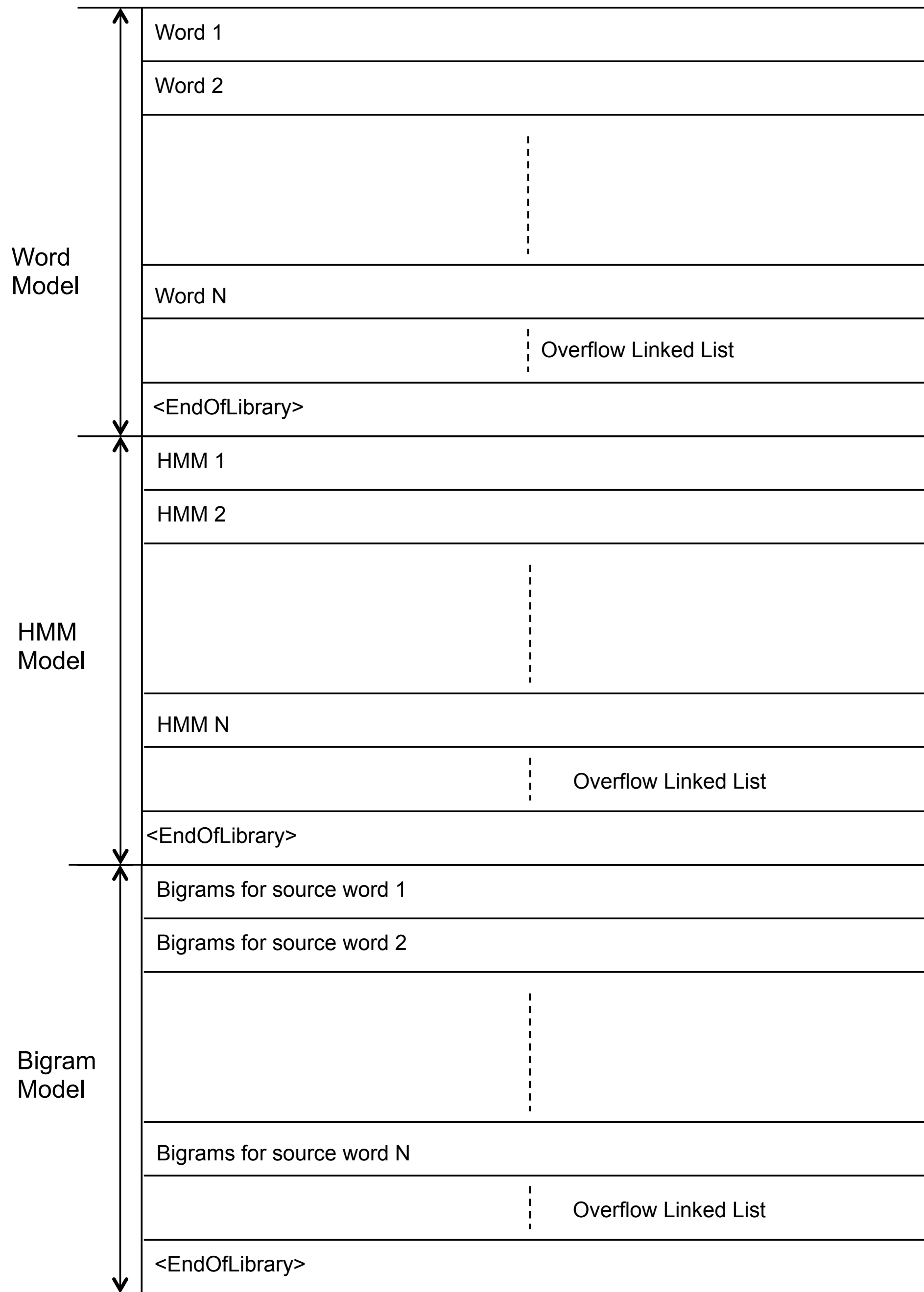


Figure 4.17: Flash Memory Format for the Viterbi Unit

Each Word Entry
<LM prob, num_HMMs, hmmID1, hmmID2 … hmmIDn> <next_ptr>

Each HMM Entry
<SenID1, tp_in, tp_self, SenID2, tp_in, tp_self …. > <next_ptr>

Each Bigram Entry
<dest word ID, LM prob, dest word ID, LM prob, ….> <next_ptr>

Figure 4.18: Storage Format for each Word and HMM

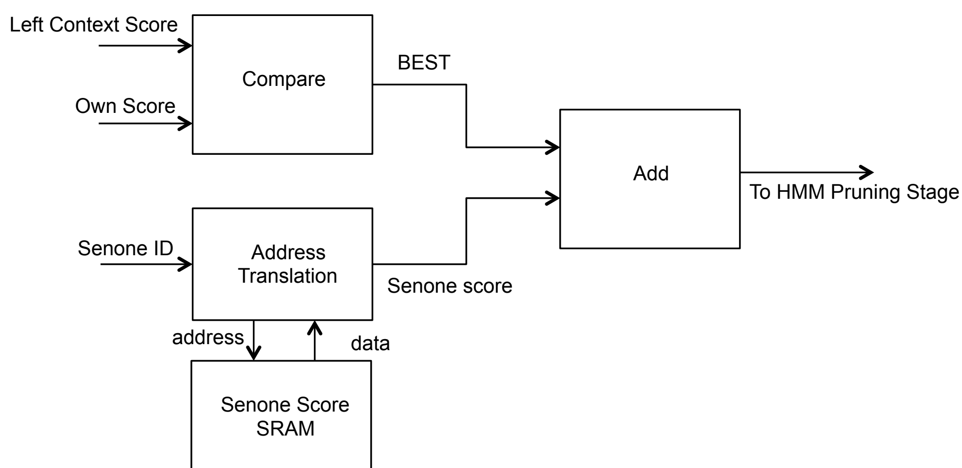


Figure 4.19: Phone Scoring Unit

N-best path pruning, we apply the same technique among multiple active list entries of the same word. Shown in Fig. 4.20 are the number of active states which pass pruning for an example utterance which lasts 381 frames.

$$T_{t+1} = T_t + \alpha(1.1 * N_{set} - N_t) \tag{4.1}$$

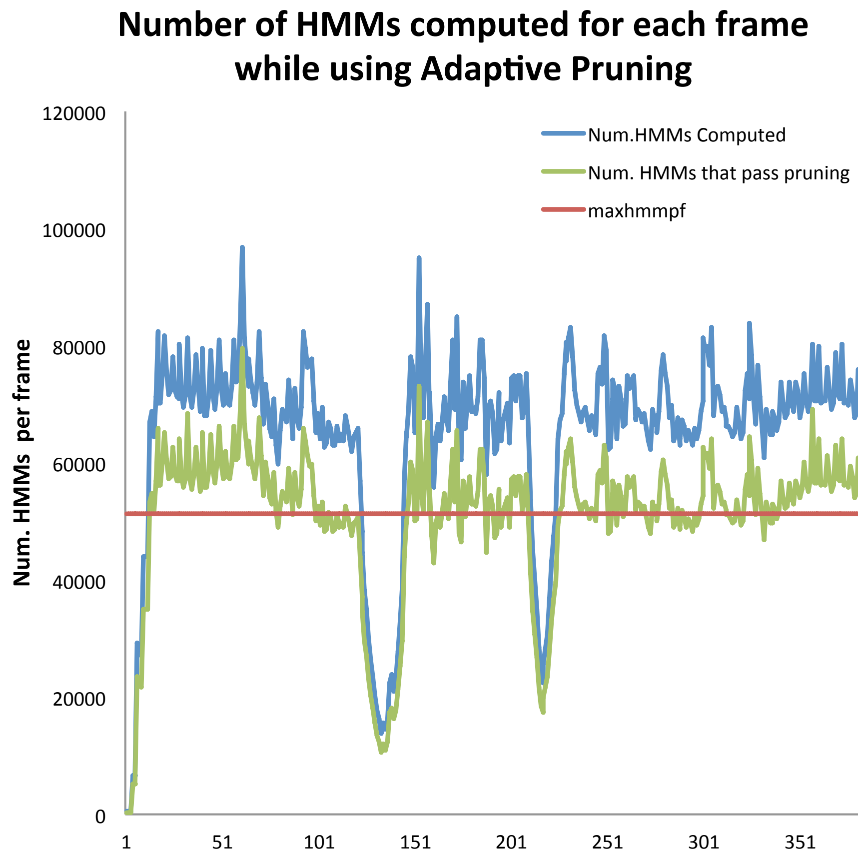


Figure 4.20: Number of Active HMMs per frame over an entire utterance which lasts 381 frames

4.7.4 Active List Generation

The active list is maintained at word level. Since we implement the word dependent N-best search algorithm, there is a separate active list entry for each combination of a word and its predecessor. We store all the entries for one word together. Each word entry contains entries for all the active HMMs within that word. For each active HMM entry, we store the scores of all states in the HMM. For each word entry, we store its Word ID, the word ID of the previous word it transitioned from, start frame of the first HMM of the word, the ID of the first HMM state in the word and the start frame of the last HMM of the word. After the HMM is scored, we compare the best of its state scores to the adaptive pruning threshold. If it passes this threshold, we add it to the active list for the next frame.

```
Each HMM Entry
<HMMID, Score1, Score2 …. ScoreN>

Each Word Entry
 < WordID, prev_best_score, nbest_pt, num_nbest, start_senID>
 < prev_WordID1, start_first_frame,  start_last_frame,
    HMM_entry_1,  ………, HMM_entry_N>

 < prev_WordID2, start_first_frame,  start_last_frame,
    HMM_entry_1,  ………, HMM_entry_N>
                          ¦
                          ¦
 < prev_WordIDn, start_first_frame,  start_last_frame,
    HMM_entry_1,  ………, HMM_entry_N>
```

Figure 4.21: Active List Format in DRAM

4.7.5 Word Lattice Generation

Whenever the last HMM of a word gets deactivated, we add it to the word lattice. For each word in the lattice, we store its word ID, previous word ID, score, start frame for the first HMM of the word, start frame for the last HMM of the word, and end frame for the last HMM of the word. This is the same format as used by sphinx and discussed in [12]. It provides both, path information and time information for each recognized word. Such a lattice can easily be converted into a word graph, N-best sentence list, N-best word list or any other lexical tree notation which is required by the software.

4.7.6 New Word and HMM activation

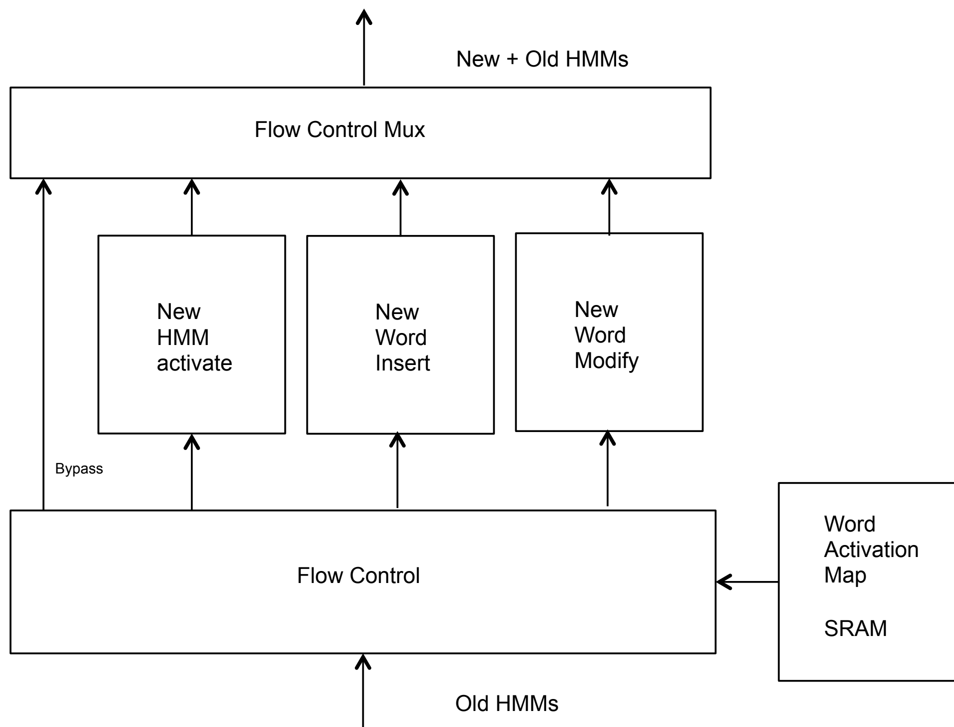


Figure 4.22: New Word and HMM Activation Block

For activation of new HMMs within a word, the score of the last state of an HMM is checked against the HMM exit threshold. If this threshold is passed, the next HMM within the word is activated.

The method for activating new words is not so straight forward. Here, we have to make a transition from a word that has exited to all possible sounds it can transition to. The words in the dictionary are grouped by starting senones, making it easy to activate a group of words starting with the same sound. We perform this task using a word activation map. A word activation map is shown in Fig. 4.23. For every word in the active list, we read the row corresponding to its starting senone from the activation map. This tells us which new entries need to be activated for this word as transitions from words inserted into the lattice. As existing word entries are read from the active list, their previous word IDs are compared against the word lattice entries pointed to by the activation map. If entries already exist, the best of the existing and new entry is chosen. After all the existing entries for a word have been processed, the remaining new entries are appended to the list.

The entries in the word dictionary are grouped by the starting senones of words. We maintain the same senone order in the word activation map. When the activation block detects that a word with a different starting senone is encountered, it appends all words for the previous starting senone from the dictionary on to the active list, before it starts processing the new set of words with the same starting senone.

4.8 Scalability of the Design

A portable speech recognition hardware implementation needs to be scalable for two reasons. One is to support larger acoustic and language models, which provide better

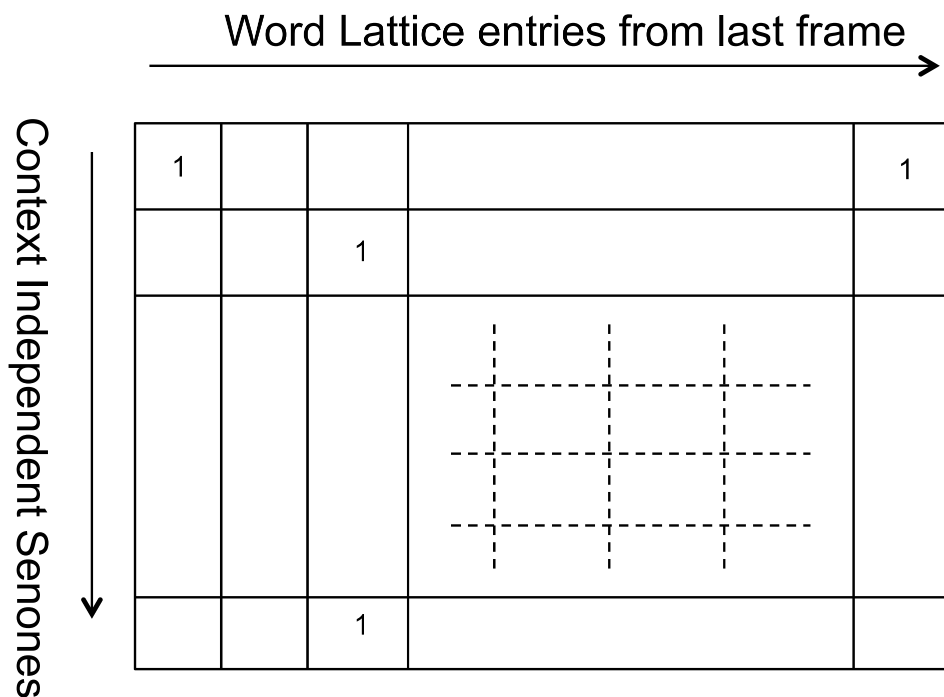


Figure 4.23: Word Activation Map

accuracy and support larger tasks. The second is to be able to decode multiple streams of speech, which would be useful in a server based speech recognition system.

Scalability for larger acoustic models can be easily acheived by adding more flash memories and distance calculation units in parallel as shown in Fig. 4.11. For larger language models, we can again use multiple flash memories and DRAMs to improve our memory bandwidth.

For supporting multiple streams of speech, we can replicate the entire senone score unit and viterbi decode unit as shown in Fig 4.24. The interface and control unit will have to be redesigned to keep track of utterance IDs for each stream so that they can be dispatched to the respective SSU and VU. Also, each command coming from software would now have an utternace ID for identification. We would also need a faster interface

like PCI express to handle the increased communication between CPU and hardware. Such a configuration is ideal for a server based speech recognition system [32] where multiple threads serve multiple users simultaneously.

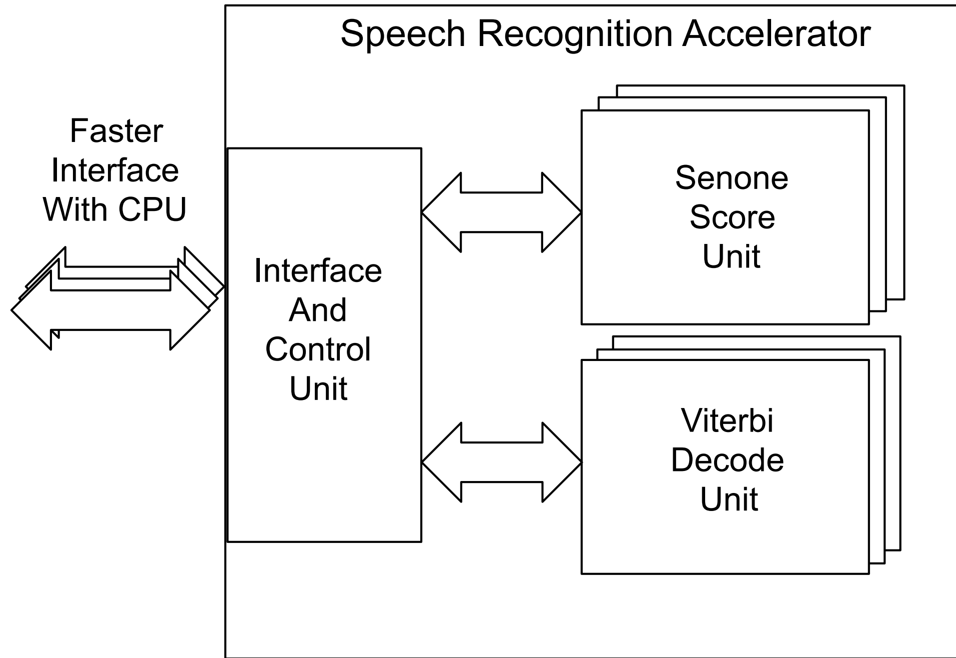


Figure 4.24: Hardware Scalability

Chapter 5

Hardware Modeling Methodology

A parametrized Transaction Level SystemC model of the proposed hardware architecture was developed. This model was used for design space exploration and for estimation of performance, area and power. The model was integrated with CMU Sphinx.

5.1 System C

SystemC [33] is a set of C++ classes and macros which provide an event-driven simulation. It allows us to simulate concurrent processes using C++ syntax. It offers many data types in addition to C++ and also user defined data types. The main application of SystemC is for System Level Modelling. It can be used to mimic hardware as it also has many similarities to VHDL and Verilog.

Our choice of language for modeling the hardware architecture is SystemC. The main advantages of SystemC are that it is object oriented and suitable for writing transaction level models. Since it can be compiled using GNU compiler, it can be easily integrated and compiled with existing software solutions. This makes it ideal for development of

hardware software co-designs. It allows us to run simulations with the software and the hardware model together and immediately see the effects of changes in hardware parameters on the overall system performance. This is very useful to carry out hardware design space exploration. It also makes it easy to model various interfaces and communication protocols.

5.2 Transaction Level Modeling (TLM 2.0)

TLM 2.0 is an industry standard for transaction level modelling [34]. It is well suited for hardware software integration and performance analysis. It defines a solid API and suggests data structures that enable model interoperability. TLM 2.0 uses two coding styles depending on the timing-to-data dependency.

5.2.1 Loosely Timed Models (LT)

These models have a loose dependency between timing and data, and are able to provide timing information and the requested data at the point when a transaction is being initiated. They do not depend on the advancement of simulation time to produce a response. Resource contention, arbitration or pipelining cannot be modelled using this model.

5.2.2 Approximately Timed Models (AT)

These models have a much stronger dependency between timing and data. They depend on events or simulation time advances to provide a result. Such a model can easily simulate pipelining of transactions or arbitration.

5.2.3 Mixed Model

For modelling our system, we use a combination of both loosely and approximated timed models. All the transactions in our model which overlap memory accesses are loosely timed, as the memory latencies are the obvious bottleneck. The pipelining and parallelism in arithmetic operations is approximately timed for performance analysis. The interface between the software and hardware is also approximately timed.

5.3 TLM vs RTL

Using a SystemC TLM 2.0 model gives us many benefits. It provides unified environment for developing fast functional models and timing accurate models. It is OS and Platform Portable. It is much faster to develop and change compared to writing RTL. Simulations can be run much faster compared to RTL, as shown in table 5.1. This result is for simulation of the distance calculation unit (FPU). The loosely timed model uses blocking transport calls while the approximately timed model uses non-blocking transport calls. RTL for the distance calculation units runs at 100MHz and has 11 pipeline stages. It is seen that SystemC provides performance benefits while accurately representing the hardware.

Table 5.1: Simulation time for SystemC TLM Models and RTL for 1 sec of real time speech

SystemC LT Model	SystemC AT Model	RTL (for FPUs)
40 sec.	13 min.	1.5 hrs.

5.4 Interaction between SystemC Model and CMU Sphinx

Sphinx was modified to send commands to the SystemC model of the hardware which runs on a separate thread. The communication between Sphinx and SystemC was done using a shared memory structure with semaphores. An example of the interaction between Sphinx and SystemC simulation threads is shown in Fig. 5.1. This allowed us to observe the effects of different hardware configurations on the end accuracy of the recognition process.

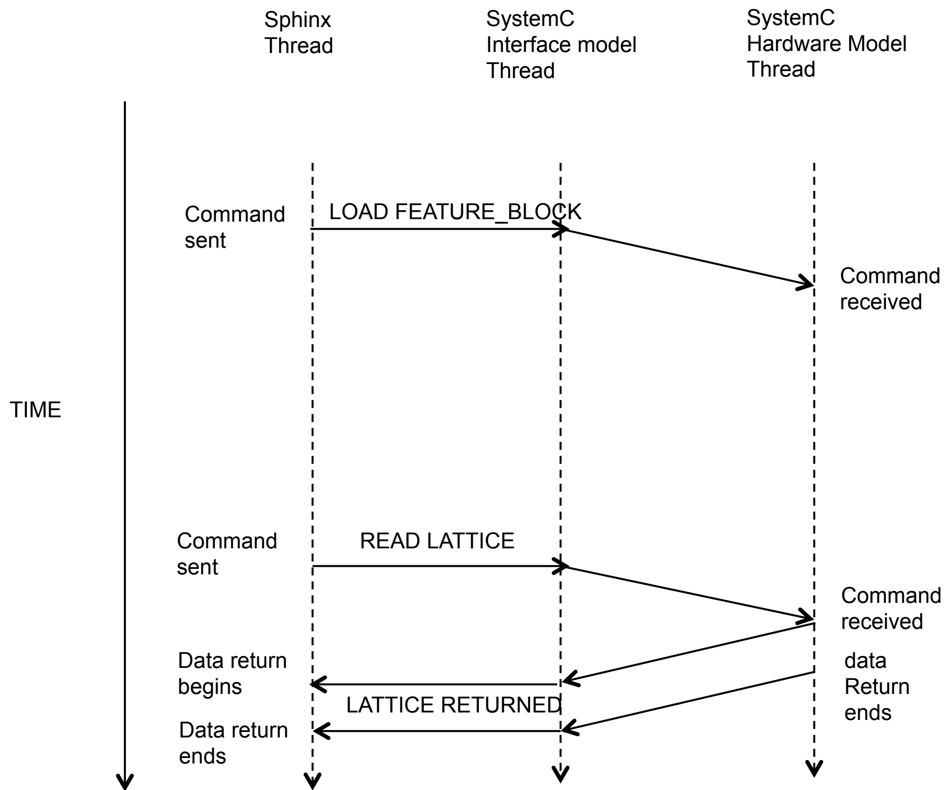


Figure 5.1: Example Simulation with SystemC

5.5 Structure of the SystemC Model

The SystemC model was transaction based at the boundary between the hardware and CPU. After the CPU writes the command and data into the shared memory, the interface delay is modeled, after which, a transaction is generated and sent to the hardware. One transaction is sent per frame. The data returned from the transaction is written back into the shared memory after modeling of the return interface delay. The CPU / software reads the data from the shared memory.

Inside the hardware model, interaction between threads, methods is done using events. Each module in the hardware is modeled as *SC_METHOD* and triggered by an event generated by the module before it. The module which model the buffers that interact with CPU are modeled as *SC_THREAD* since they constantly keep checking for new available data. These buffers can be updated or read by the CPU at any time during the course of a transaction.

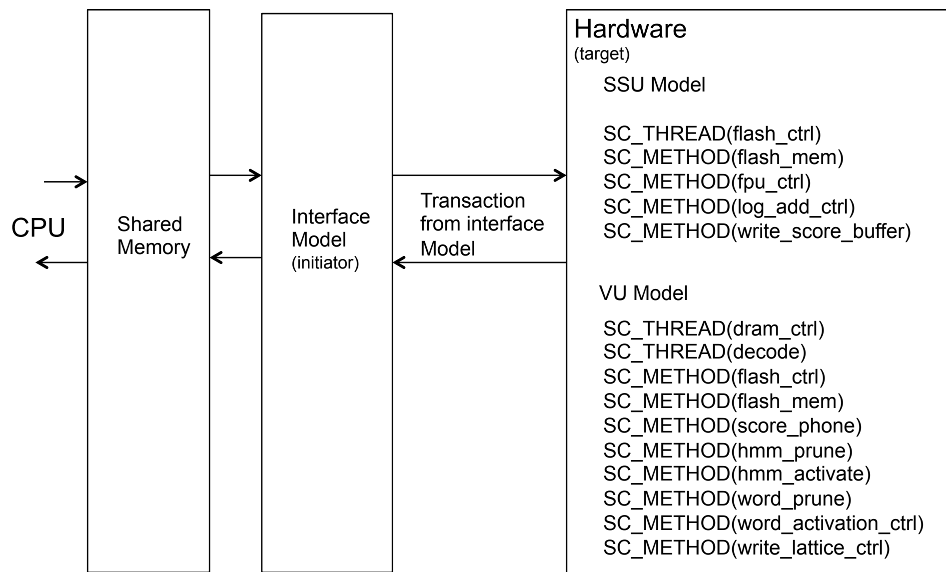


Figure 5.2: Structure of the SystemC Model

Fig. 5.2 shows the different functions in the SystemC Model. The functions which fetch data that needs to be processed, are modeled as threads. In the SSU model, *flash_ctrl*() loops through all the senone IDs in the acoustic library and fetches the Gaussian Mixtures for each senone. Once its done with the current frame, it waits for the next *LOAD_FEATURE_BLOCK* command. Once data is returned by *flash_mem*() the rest of the methods are called to process the data and write the senone scores into an SRAM. In the VU Model, the *dram_ctrl*() thread waits for the new frame event. As soon as a new frame begins, it starts reading the active list from DRAM. The HMMs read from the list are processed by the *hmm_scoring*() method. The *flash_ctrl*() and *flash_mem*() threads model the flash memory access for the tri phone and word dictionary. The *adaptive_pruning*() method prunes out tri-phones which are below the beam threshold and *hmm_activation*() and *word_activation*() carry out intra and inter word transitions. Any word which is recognized, is added to the word lattice along with its predecessor and time stamp by *write_lattice_buffer*().

5.6 Transactions, Transport calls and Event Generation

The interface model generates the payload of the SystemC transaction from the data written into the shard memory, shown in Fig. 5.2. The transaction supports all commands and data discussed in Section 4.4. Once this transaction is received, the top level thread in the hardware model interprets the command and data, populates the input data buffers (modeled as queues) and generated the appropriate events needed to kick of the threads that start processing of the incoming software command.

5.7 SystemC Behavioral Model of the Hardware

In this section, we will discuss the SystemC functions which have been used to model hardware behavior. The synchronization between these functions is done using events. Initially, we had transactions going from each module to another. However, we soon realized that we needed only one transaction at top level, coming from the CPU interface to the hardware. The rest of the system could be adequately modeled using events. Below, we will discuss each of the behavioral functions briefly.

5.7.1 Utterance Initialization

The initialize() function is called to initialize the entire hardware for a speech recognition routine. This function is also called at the beginning of every new utterance. It sets the initial beam pruning threshold for the search in the ADAPTIVE_PRUNING hardware block and resets the size of the active list to zero.

```
initialize()
{
   //initialize utterance dependent parameters and variables
   //reset adaptive pruning threshold
   //set initial pruning threshold
   //reset active list
}
```

5.7.2 Frame Initialization

The frame_begin() routine is called every frame. Here, we increment the frame number, select offsets for active lists in the NEW_HMM_WORD_ACTIVATION hardware block

and word lattice for the new frame in the LATTICE_CONTROL hardware block. We also update the previous best score in the ADAPTIVE_PRUNING hardware block which is used for pruning and calculate the adaptive pruning threshold based on the formula discussed in section 4.7.3.

```
ssu_frame_begin()
{
   //initialize frame dependent parameters and variables
   //calculate adaptive pruning threshold
   //read feature vector from senone incrementer block
}
```

5.7.3 Flash Control for SSU

This function gets the senone ID from the senone incrementer block. This senone ID can be converted into the address for the Flash Memory and issues the first read for the senone. The first read returned has a field for length which is in turn used to achieve further reads.

```
ssu_flash_ctrl()
{
   // read senone ID from senone incrementer block
   // translate senone ID and acoustic library offset into
   // starting flash address
   // issue flash read
   // notify fpu control  when flash read data is available
   // issue reads for the senone until all senone data is read
```

```
   // read next senone from incrementer block

}
```

5.7.4 Distance Calculation

This function calculates the inner summation for equation 2.4. This function is untimed in the SystemC model as the data path never gets starved. The SSU has no dependency on other operations and never stalls once started.

```
ssu_fpu_ctrl()
{
   // interpret data from the flash read
   // store mixture weight and log reciprocal
   // perform distance calculation on feature vectors using
   // gaussian data from current read and subsequent reads
   // pass the accumulated result to scale_and_Weight module
}
```

5.7.5 Adding Mixture Weight and changing Log Base

After the inner summation, this function scales the result using the mixture weight parameter in the acoustic model and changes the base of the logarithm to 1.0003. This function is performed in the Distance Calculation hardware block.

```
ssu_scale_and_weight()
{
   // add mixture weight and log reciprocal stored
```

```
   // by the fpu_ctrl module from the first flash read of the senone
   // pass the result to log add control
}
```

5.7.6 Logarithmic Addition

Since the entire senone scoring calculation is performed in the logarithmic domain, the outer summation becomes a logarithmic addition. Here, we estimate the value of $log(A+B)$ from $log(A)$ and $log(B)$. This function performs this operation using a look up table. This stage involves an SRAM access. However, since there is not case of a stall, we again do not need a complicated timing model for this.

```
ssu_logadd_ctrl()
{
   // perform logarithmic addition on gaussian results
   // received from the scale_and_weight unit, for the same senone
   // write accumulated result to senone_score_buffer when all
   // mixtures for the senone are done
}
```

5.7.7 Memory Model

The Flash Memory Model stores the data in a character array. This is a byte accessible array. This function implements the address translation logic for the memory and reads the required chunks of bytes from the array, for the address received. It then models the read access time on the backward transport call to the memory read transaction and returns the read data.

```
ssu_flash_mem()
{
   // receive flash address from the flash_ctrl module
   // read data from the memory array after decoding the flash address
   // model the read access time for the flash memory
   // return read data to flash_ctrl module
}
```

5.7.8 Viterbi Decoder Top Level Function

This top level function passes data between hardware blocks in VU (see Fig. 4.15). This function gets an HMM from the word / HMM activation function. It determines if this is a start of a new word and issues a word read to the flash_ctrl function. Then it issues HMM dictionary reads to the flash_ctrl function. On receiving the read data, it forms a data packet for the phone score blocks and passes them through the pipeline. This function represents the top level functions and glue logic for the VU.

```
vu_decode()
{
   // receive active word from dram ctrl block
   // issue word structure read to flash_ctrl
   // issue triphone read to flash_ctrl for all active triphones
   // pass read data to multiple parallel score blocks
}
```

5.7.9 Flash Control for VU

This function receives either a bigram read, word model read or a triphone model read. For the read, it translates the word ID or triphone ID into the NOR Flash address and accesses the memory. It stores the returned data in an SRAM buffer. For subsequent reads in the same word, it returns the data from the SRAM buffer if it is already present there. At the beginning of a new word, it purges all the SRAM buffer data for the previous word and uses it for the new word. We do not cache triphones across word boundaries. For Bigrams, we prefetch rows for the Flash into an SRAM. The SRAM is partitioned logically to support multiple rows of bigrams of a single source word.

```
vu_flash_ctrl()
{
   CASE 1:
   // receive word ID & word read command from decode block
   // translate word ID to flash address for word structure
   // issue flash read
   CASE 2:
   // receive word ID, triphone ID & triphone read command from decode block
   // translate word ID  , triphone ID to flash address for triphone structure
   // if SRAM buffer bit not valid then
   // issue flash read
   CASE 3:
   // receive source word ID & bigram pointer:
   // translate to Bigram address
   // issue flash read
```

```
}
```

5.7.10 DRAM Controller

This function is a wrapper which send requests to the DRAMSim2 controller and receives data back from it. The DRAMSim2 simulator models the DRAM delay and passes the data back on to the DRAM controller. Our DRAM controller is not very complex. It reads one page from the DRAM active list at a time and then writes one page of the active list for next frame back into the DRAM. It thus alternates between pages of read and write.

```
vu_dram_ctrl()
{
   // from active list start address to end address
   // keep reading from dram as long as there is not
   // pipeline stall
   // pass the data read from dram to decode unit
}
```

5.7.11 Next Word / HMM Activation Block

This function uses the word activation map and activates new words and HMMs (using the vu_hmm_activate() function) of modifies existing entries if a combination of word and predecessor already exists in the active list.

```
vu_word_activation_ctrl()
{
```

```
   // if word bit is high in new word activation map
   // and if word is not already active with a last frame
   // word lattice list entry as predecessor
   // then  activate word and send it to decode function
}
```

5.7.12 Fetching Senone Scores for Triphone Scoring

This function uses the senone ID and current frame number and translates this into an address to read senone scores from the senone score SRAM. On doing this, it sends that senone with its senone score, state score, left context score and transition probabilities to the phone scoring unit. This is part of top level logic in the viterbi decoder.

```
vu_read_senone_score()
{
   // receive senone ID from score_phone module
   // translate senone ID into buffer address
   // return senone score to score_phone module
}
```

5.7.13 Scoring each phone in a triphone

This function uses equation 2.7 to calculate the new state score for the senone. This models the HMM_SCORING hardware block.

```
vu_score_phone()
{
   // issue senone score read to read_senone_score module
```

```
    // compare last frame state score of itself and left context
    // add transition probability and senone score
}
```

5.7.14 HMM Pruning

This function works on HMM level and not at senone level. It finds the best score among the scores of all the senones in the triphone. This score is then compared to the pruning threshold which is calculated by the vu_adaptive_pruning() function. If the HMM does not pass pruning, it is discarded and not written back to the active list. This is a part of the ADAPTIVE_PRUNING hardware block.

```
vu_hmm_prune()
{
    // get best state score from the triphone
    // compare best triphone score to pruning threshold
    // if triphone score is better than pruning threshold
    // keep the triphone active
    // else deactivate (discard) the triphone
}
```

5.7.15 HMM Propagation

This function uses the pruning threshold to propagate the HMM to the right of the current HMM in a word. For this, it compares the score of the right most (last) state of the HMM under consideration and compares it to the HMM pruning threshold. If it passes the threshold, the next HMM is activated by inserting it to the active list after

the current HMM. This is a part of the NEW_HMM_ACTIVATION block in hardware.

```
vu_hmm_activate()
{
   // get state score from last state of the triphone
   // compare last state triphone score to pruning threshold
   // if  score is better than pruning threshold
   // activate right context triphone if not active already
   // else do nothing
}
```

5.7.16 Word Pruning based on word threshold

Once the right most state of the last HMM in a word has passed pruning, this function compares it to a more stringent word pruning threshold. If this threshold does not pass, the word is deactivated. Also, a word is deactivated if none of the HMMs in the word pass HMM threshold. This is a part of the ADAPTIVE_PRUNING hardware block.

```
vu_word_prune()
{
   // if triphone is last triphone of the word
   // compare last state triphone score to pruning threshold
   // if last state score is better than pruning threshold
   // mark word as possible exit
}
```

5.7.17 Adding recognized words to the output lattice

If a word has passed the word pruning stage and been declared as identified, this function populates the word lattice entry from the active list entry for the word and writes it into the buffer. If the buffer is full, this function stalls the VU pipeline till the buffer is emptied. This function models the WORD_LATTICE_CONTROL hardware block.

```
vu_word_lattice_ctrl()
{
  // increment lattice write pointer,
  // write lattice entry
}
```

5.7.18 Calculation of Adaptive Pruning Threshold

This function uses a simple linear approximation of the adaptive pruning technique from [13] to calculate the pruning threshold. This is performed at frame initialization in the Adaptive Pruning Block.

```
vu_adaptive_threshold()
{
   //calculate the adaptive pruning threshold
   //based on the formula for adaptive pruning
   //if this threshold is worse than the initial
   //beam pruning threshold, select the initial
   //threshold as the new threshold
   //If NT < 1.1*NSET, use initial threshold
```

```
}
```

5.7.19 SPI interface specific functionality

This function models the interface delays for the spi interface. It polls the shared memory structure between sphinx and the SystemC hardware model to check for new commands and accordingly models the interface delays.

```
spi_ctrl()
{
   // poll to check if software (testbench or sphinx)
   // has written to shared memory
   // if new spi command has been received,
   // interpret the written data to calculate the total
   // data transferred for that command
   // model interface delay
   // then generate systemC event to notify hardware of new command

   // if data has to be returned to CPU (testbench or sphinx)
   // hardware systemC thread will write to output buffer
   // interpret this written data to calculate bytes transferred
   // model interface delay
   // write this data to shared memory
}
```

5.7.20 Interpretation of commands from CPU

This function has a case statement for the commands received from the SPI interface. Within each case, there are calls to appropriate methods which are needed to service the commands. This models the Interface and Control Unit in hardware.

```
command_ctrl()

{

//decode a command  received from spi_ctrl()

//generate events to invoke appropriate modules

//read data from buffers and return it to spi if needed

}
```

5.7.21 Top level wire connections and signals

These are top level events which tie up the SSU, VU and interface control units in the hardware accelerator.

```
top_level_ctrl()

{

//glue logic between modules

}
```

5.8 Estimation of Timing Parameters for the SystemC Model

Parameters in the SystemC Model which are used to simulate the performance of the system need to be accurately estimated in order to get results from SystemC simulations,

which will be close to the actual hardware. We used results from synthesized verilog to estimate these parameters. The major operations involved in the entire algorithm were identified. These were the functions from the SystemC model which describe the behavioral functionality. These operations we further subdivided into basic arithmetic operations on data and control flow operations. The latency for each of the arithmetic operations was estimated by synthesizing Synopsys Designware components which support these operations. Where ever needed, these components were stitched together using glue logic, to imitate the control flow signals. These were verilog case statements which synthesize into a multiplexer or decoder in most cases. If individual arithmetic operations needed to be pipelined further and a pipelined designware component was not available; a guesstimate was made by dividing the latency with number of pipelining stages and adding a flip flop setup time and clock to Q delay per pipeline stage. The worst case latency from all these blocks was used as the system clock with an additional 15% added to get a conservative estimate.

5.9 DRAM performance estimation

DRAM performance was estimated using DRAMSIM2 Memory System Simulator [35] from University of Maryland. It is a DDR2 / DDR3 memory system model which can be used for system and trace based simulations. An accurate DRAM simulation is essential to the analysis of a speech recognition system as memory is the bottleneck for this application, especially the speech decoding algorithm. DRAMsim2 is good at modeling variable latency that can be caused by access to different banks of the DRAM, read/write interleaving, etc. This simulator has a $MemorySystem$ object [35] wrapper which provides a simple user interface. This object can be instantiated in any C/C++

code. Requests can be sent to this object. The simulator can be configured using two ini files. The device ini file has parameters for the specific DRAM device that we want to model. This can include various timing constraints or power parameters. In our simulation, all these parameters were filled in from the manufacturer data sheet. The simulator package also comes with some default ini files with Micron DDR2 devices. The other ini file that needs to be configured is the system ini file. This file has parameters independent of the DRAM device. This includes number of ranks, address mapping, debug options for the simulator, memory controller queues, etc.

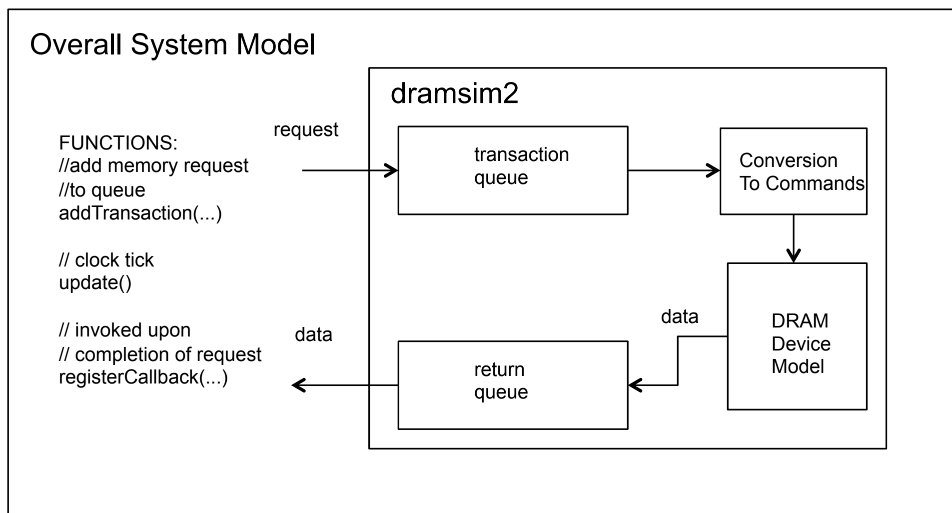


Figure 5.3: DRAMsim2 Usage

5.10 Architectural Exploration using SystemC Model

The SystemC model was used for architectural exploration. This included analyzing the effect of parallelism, pipelining, memory bandwidth and memory organization on the

overall performance of the system. Each of these components in the system was parameterized in the SystemC Model. The overall system performance was plotted against different settings of these parameters.

Shown in Fig. 5.4, are the simulations for the SSU. The entire operation was divided into basic block functions and the latency of each function was obtained by synthesizing the required design ware components in the Spansion CS239LS 180nm Standard Cell Library. The SystemC model was designed to automatically model the pipeline stages using these numbers and the flip flop delays in the library. We can clearly see that parallelism of at least 4 is required to obtain the best possible performance in a reasonable number of pipeline stages. The required parallelism depends upon the memory bandwidth available for reading the acoustic model. The memory bandwidth governs the number of operands available to the functional units in one NOR Flash memory read. The parallelism or pipelining should to be high enough to consume all the data in the memory read, so that the computation / data path is not the bottleneck for the hardware. It is observed that after a certain point, increase parallelism does not help as we have reached the memory bandwidth limitation, beyond which data is not being read fast enough to keep all the functional units busy. We choose this knee of the curve as our design point for the SSU.

The Viterbi Unit (VU) is not computationally intensive and does not use any parallelism. Hence, the number of pipeline stages in governed by the clock of 100MHz and the number of operations in the data path which need to be registered. We have a total 22 pipeline stages in the VU.

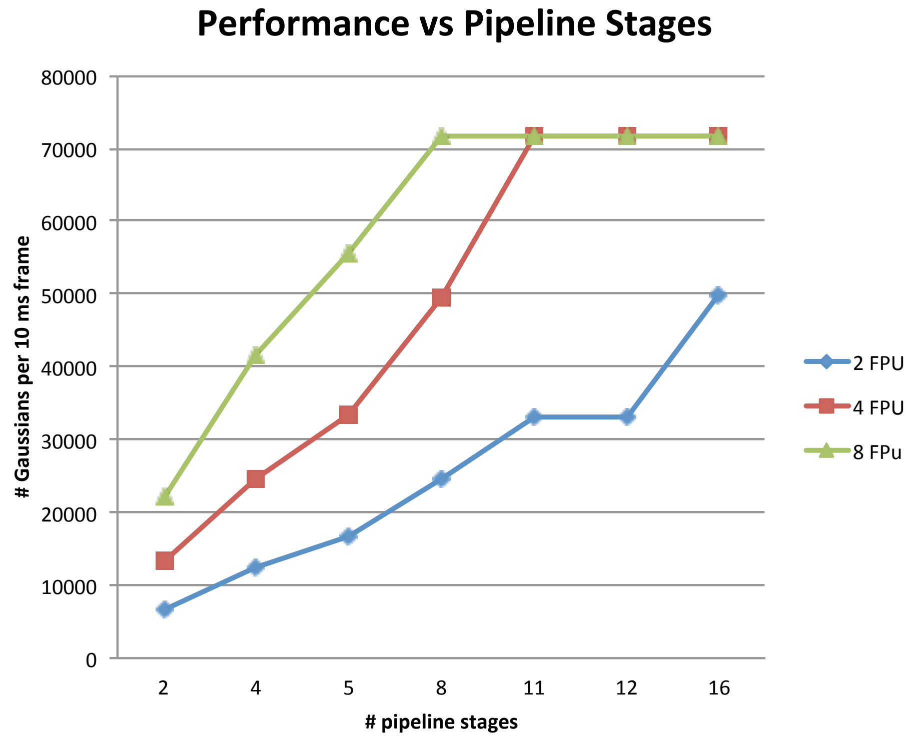


Figure 5.4: SystemC Simulation to explore effect of parallelism and pipelining on overall SSU performance

5.11 Area and Power Estimation

For area and power estimation, a combination of System C and RTL was used [36]. The basic building blocks of the hardware like adders, multipliers etc. were identified and synthesized using design-ware components and some glue logic in Spansion CS239LS 180nm Standard Cell Library. The total number of such building blocks which would be required for each operation modeled in SystemC was estimated. The total logic area was obtained by hand calculations and scaled by a factor of 1.2 to account for inconsistencies. Area for SRAM and NOR Flash was obtained from Spansion, LLC. The area for off chip DRAM was obtained from manufacturer specifications.

For calculating the power, counters were used in the SystemC model to count the number of arithmetic operations on each basic building block and memory read/write operations. The power consumption for each basic block was obtained from synthesis, assuming 50% toggling of input. For the SRAM and Flash, the read/write power was obtained from Spansion, LLC. The time for which the memory was active was calculated from simulations to get the total power consumption. The total power was scaled by a factor of 1.2 to account for inconsistencies.

5.11.1 DRAM Power Estimation

For DRAM, we used the methodology specified in [37] along with current and voltage values from the manufacturer data sheet. Equations 5.2 through 5.7 used for calculating the power consumed by the DRAM. Equations 5.8 and 5.9 show the scaling of power with operating voltage and operating frequency. All the current parameters in these equations can be obtained from the manufacturer's data sheet for the device being used. The values for number of activates, reads and writes were obtained from our SystemC simulation

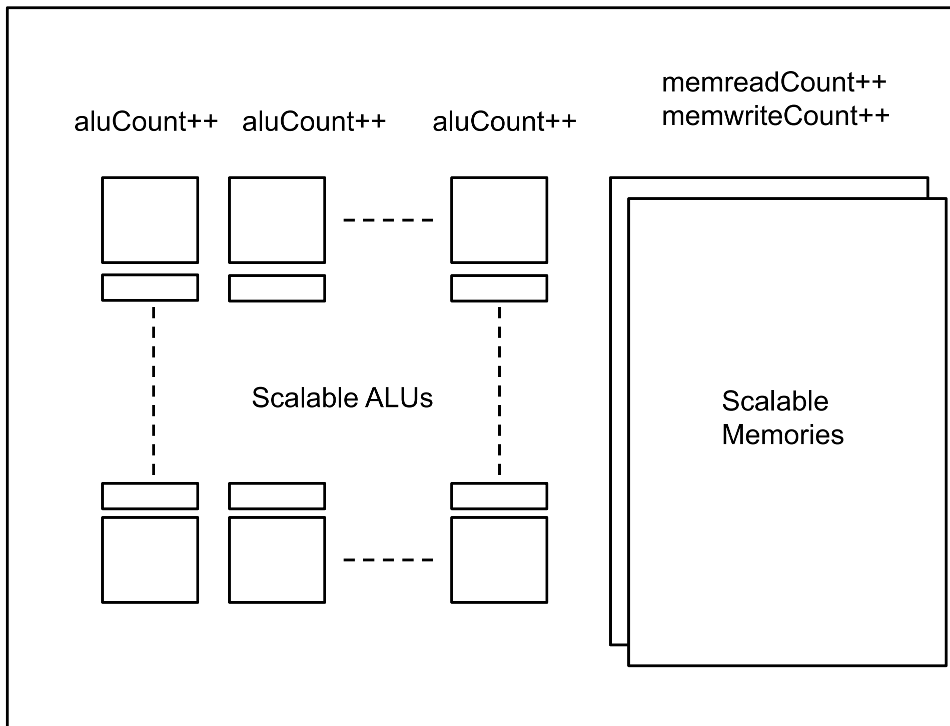


Figure 5.5: Counters for Power Estimation in the Hardware SystemC Model

along with DRAMsim2 [35] as explained in section 5.9. A more detailed explanation of all these equations can be found in [37]. For the DRAM controller power, we used the synthesis power of a simplified version of the DRAM controller we build in a previous work [38].

$$P(ACT_STBY) = Idd3N * V_{dd} \tag{5.1}$$

$$P(ACT) = (Idd0 - Idd3N) * \frac{T_{RC(spec))}}{N_{ACT} * T_{CLK}} * V_{dd} \tag{5.2}$$

$$P(WR) = (Idd4W - Idd3N) * \frac{num_write_cycles}{N_{ACT}} * V_{dd} \tag{5.3}$$

$$P(RD) = (Idd4R - Idd3N) * \frac{num_read_cycles}{N_{ACT}} * V_{dd} \tag{5.4}$$

$$P(perDQ) = V_{out} * I_{out} \tag{5.5}$$

$$P(DQ) = P(perDQ) * (num_DQ + num_DQS) * \frac{num_read_cycles}{N_{ACT}} \tag{5.6}$$

$$P(TOT) = P(ACT) + P(WR) + P(RD) + P(DQ) + P(ACT_STBY) \tag{5.7}$$

$$P(new_Vdd) = P(spec_Vdd) * \frac{(new_Vdd)^2}{(spec_Vdd)^2} \tag{5.8}$$

$$P(new_freq) = P(spec_freq) * \frac{(new_freq)}{(spec_freq)} \tag{5.9}$$

5.12 Communication Overhead Modeling

We chose a 50 Mhz Quad Serial Peripheral interface to communicate between hardware and CPU. This interface was also modeled at transaction level in SystemC. The thread running the interface model shown in Fig. 5.1 constantly polls the shared memory for updated data. Whenever the Sphinx writes a command into the shared memory, the interface thread models the interface delay for this command and accompanying data, after which it generates an event which tells the hardware model that a command has been received.

5.12.1 Serial Peripheral Interface (SPI)

The serial peripheral interface (SPI) bus is a synchronous data link bus. It can operate in full duplex mode. For this purpose, it has separate pins for data in and data out. However, the Quad I/O pins are bi-directional and data can be sent only in one direction at one time. In this system, the CPU is the master and the hardware accelerator is the slave. There is no acknowledgement protocol to confirm the receipt of data at either the master or the slave. For this design, we observed that the data rates offered by a simplistic Quad pin SPI interface are enough to transfer the required data to and from the hardware to CPU. Hence, we did not see the need for a full parallel interface like I2C or other parallel peripheral buses. We chose to use a bus like SPI which has low pin count.

5.13 Validation of SystemC Model

Validation of the SystemC model functionality was done using a modified version of the CMU Sphinx [14] code. The SystemC model was first written to implement a traditional time synchronous viterbi beam search similar sphinx. The Sphinx code was modified to print out a word lattice in a format similar to the one proposed here. A test data set (400 sentences from WSJ0 corpus) with the same language model (64K bigram trained on WSJ0 corpus) were simulated. The output word lattices of SystemC and Sphinx were then compared to ensure functional correctness. The SystemC code was further modified to implement the proposed N-best algorithm. The implementation was validated using the condition that the word lattice obtained in the sphinx algorithm was always a subset of the new word lattice. The criteria used to validate the implemented algorithm is *Lattice Error Rate* [12]. This is the lower bound word error rate from the lattice. In other words, it is the word error rate we get if we choose the lattice path (hypothesis) with the lowest word error rate. This criteria is also called an oracle error rate since it needs perfect knowledge of which path to pick. We ensured that the lattice error rate of the proposed implementation met the lattice error rate of the exact N-best implementation by widening the beam width and reducing the tri phone and word pruning thresholds.

To validate the functional correctness of the senone scoring and viterbi decoding equations, a trace based simulation was used. The input traces (operands) for both the modules were generated for two utterances from the WSJ0 corpus. These traces were run through both Sphinx and the SystemC Model and outputs were compared using an automated script.

5.14 Word Error Rate Calculation

For calculation of the end word error rate of the system, NIST Speech Recognition Scoring toolkit (SCTK) [39] was used. This tool kit can be used to calculate the word error rates for a batch of utterances. It needs to be supplied a text tile with expected results (reference file) and actual results (hypothesis file). Utterances need to be delimited by a new line character. The tool kit prints out many criteria which are used in speech recognition, including word error rate, total words recognized, words recognized correctly, words inserted, words deleted and word substituted. For calculation of the lattice error rate, we used the same toolkit but modified the reference and hypothesis files to contain patch in the word lattice, delimited by new line characters. Each path was represented by a source and destination word. The reference file was obtained from sphinx and the hypothesis file from the SystemC model. The correctness criteria for the hardware was that words can be inserted in to the word lattice but not deleted or substituted.

Chapter 6

Results

In this chapter, we discuss results obtained by simulating the proposed hardware architecture using SystemC. Initially simulations were run for design space exploration to find the optimal design point. This was done using an untimed SystemC model for the multipass N-best search, as we concentrated on the end accuracy of the speech recognition system for this exercise. After selection of the optimal design point, a hardware architecture was developed to meet this accuracy in real time. The experiments were carried out on 400 sentences from wall street journal CSR I corpus.

6.1 Accuracy

First, the effect of Number of Gaussian Mixtures in the Acoustic Model on the WER of the system was explored. We chose an 8000 senone acoustic model from [40] as this was the only open source model available with gaussian mixture ranging from 2 to 64, which provided us with enough data points to run this parameter sweep. We found that the WER gets better with increased number of gaussians, which is an expected results.

For our system which targets large vocabulary, we found that the WER stagnated at 8 gaussian. Hence we chose this as our design point.

Table 6.1: Effect of Gaussian Mixture on WER using Multipass Decode

Number of Mixtures	2	4	8	16	32	64
5K Words	23.3%	22.1%	21.7%	21.7%	20.5%	20.3%
20K Words	13.9%	13.6%	13.4%	12.5%	12.4%	12.4%
64K Words	12.5%	10.8%	10.1%	10.1%	10.1%	10.1%

The second experiment we ran was the observe the effect of using a simple language model like unigram/bigram, on the WER of the system. For this simulation, we used multiple language models in the first pass of decode and the same trigram language model in the second pass. it was observed that using the unigram/bigram model certainly affects accuracy. However, it was found that increasing the beam width of the first pass (with N-best search) made sure that the best hypothesis was always included in the N-best lattice output of the first pass. This made sure that the second pass of search which used the trigram model was always able to find the best hypothesis in the second pass and maintain the end accuracy of the system.

Table 6.2: WER with N-best Bigram Pass I and 1-best Trigram Pass II

NB + 1T	1T+1T	NB+1T (wider beam)
12.6%	10.1%	10.1%

6.2 Estimated Performance, Area and Power

Our design is estimated to run at a clock speed of 100MHz, using Spansion CS239LS 180nm standard cell library and 65nm technology for the NOR Flash Memory. The SSU supports processing of an Acoustic Library of 8000 senones, 8 Gaussian Mixtures and a 39 dimensional Feature Vector in 0.85x real time. It uses a 768 bit wide flash memory to store the acoustic model. The Viterbi unit supports an N-best search using an unigram/bigram language model of 64K words. We achieve an average 0.78x real time performance. However, this number varies largely with input. We got numbers as high as 0.92x for one of the utterances in our test data. The VU uses a 256 bit wide flash memory to store word and triphone models. The flash memories have a random read latency of 80ns. We need an on-chip SRAM of 14KB for DRAM read / write buffers, log-add look up tables, HMM dictionary cache, bigram prefetch cache and word activation map. The SRAM requirement for storing senone scores and word lattice depends on the size of acoustic and word models. In our case, we need 64KB for senone scores and 16KB for the word lattice. The estimated area for SSU logic is 200K gate equivalents and for VU logic is 120K gate equivalents.

The SSU consumes an estimated 422mW in the data path and 384mW for memory reads. The VU consumes 230mW in the data path and 359mW for memory reads. The total estimated SRAM power consumption for SSU and VU is 143mW.

Table 6.3 shows the power consumed by different stages of the recognition algorithm. We can see that senone scoring is the most power hungry as the entire acoustic model needs to be read every frame. The power consumption for other stages can be reduced by caching and using a NOR flash memory which provides high access efficiency.

The proposed hardware architecture provides high memory access efficiency by using

Table 6.3: Power consumption itemized by task

Task	**Power Consumption**
Senone Scoring	867 mW
HMM Scoring	512 mW
HMM Transitions	138 mW
Word Transitions	206 mW

NOR Flash for random accesses (see Table 6.4). Our experiments indicate that the random nature of word and HMM dictionary reads greatly reduces memory access efficiency when stored in a DRAM. The memory read power for VU is greatly reduced by caching HMM dictionary reads on an SRAM till all entries or a single word have been processed. We achieved an average cache hit rate of 65.74% for HMM dictionary reads. The second major factor in improving the performance and power efficiency of a VU is the use of a simplified Unigram/Bigram language model.

Table 6.4: Memory Access Efficiency for different Hardware configurations

Configuration	Efficiency
SSU with DRAM	78%
SSU with Flash	94%
VU with DRAM	48%
VU with Flash + DRAM	89%

The hardware provides a 4.3X performance improvement (see Fig. 6.1) compared to Sphinx running on an Intel Core 2 duo 2.4GHz processor with a 4 GB 667MHz DDR2 SDRAM, while consuming an estimated 1.72W. The proposed hardware / software design partition which uses Multipass Decoding with N-best Search provides us with a generic hardware architecture which can be used with multiple front ends and multiple N-gram

language models. This architecture is highly scalable for both embedded and server implementations.

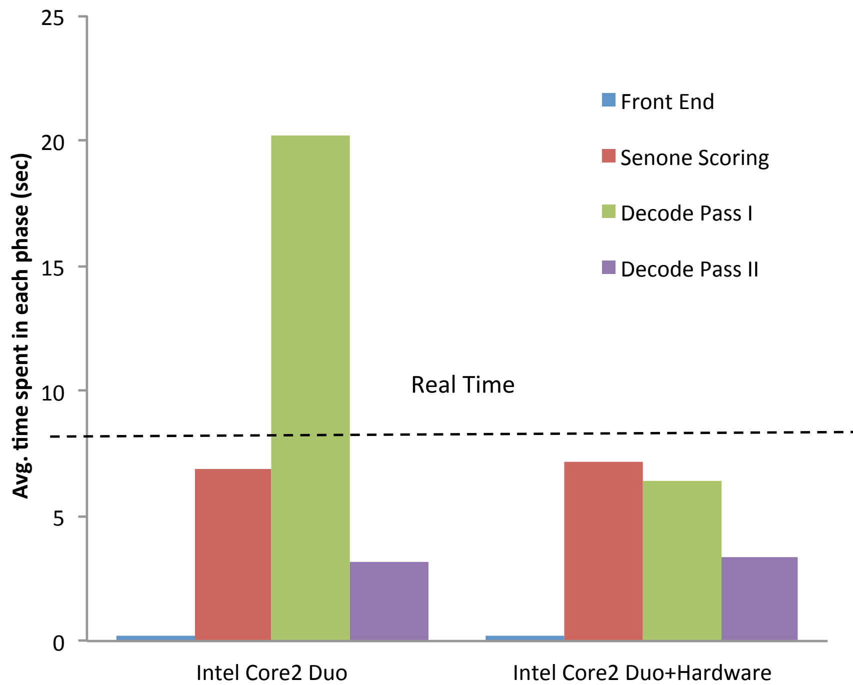


Figure 6.1: Time taken by each operation in Multi-Pass Decode on Intel Core2Duo 2.4 GHz with 4GB 667MHz DDR2 SDRAM; with and without the Proposed Hardware Accelerator (400 utterances, WSJ0 CSR I corpus)

6.3 Comparison to other work

A comparison of with previous work is provided in tables 6.5 and 6.6. [1, 20, 26] implement only the observation probability calculation unit in hardware are hence are only compared

to the SSU. [23] and [6] implement the viterbi search hence their power consumption for the search portion of the algorithm is compared to the VU. The other hardware implementations discussed in chapter 3 either do not report power consumption or ignore power consumption for memory accesses. All of these implementations concentrate on performance. All the designs meet the real time performance criteria. It should however be noted that none of these implementations support an N-best search. The figure of merit we use for comparison is the power consumed per gaussian mixture for the SSU and the power consumed per word for the VU.

Table 6.5: SSU Comparison with Related Work

	[1]	[20]	[26]	This Work
Technology	250nm	180nm	130nm	180nm
Data Format	24 bit	32 bit	32 bit	32 bit
Feature Dimensions	39	39	27	39
Num. Senones	6000	6000	358	8000
Num. Gaussians	8	8	3	8
Total Power	1.8W	735mW	15.2mW	867mW (est.)
Power Per Mixture	961nW	393nW	524nW	347nW

Table 6.6: VU Comparison with Related Work

	[23]	[3]	This Work
Technology	90nm	180nm	180nm
Data Format	32 bit	32 bit	32 bit
Num. Words	5000	20000	64000
Language Model	Trigram	Trigram	Bigram
N-best?	No	No	Yes
Output Format	Lattice	Lattice	Lattice
Total Power	196mW	3.5W	852mW (est.)
Power / Word	39.2uW	175uW	13.3uW

Chapter 7

Conclusion

7.1 Summary

In this work, we have attempted to come up with a portable hardware architecture for speech recognition. Our goal was to keep the performance of exiting ad-hoc hardware solutions while trying to make the hardware architecture generic and scalable. This involved accelerating the observation probability calculation stage of the algorithm while supporting multiple front ends. This process was relatively straight forward as this process is well suited a SIMD like architecture. For the acceleration of the search phase of the algorithm, one of the biggest challenges was to decouple the hardware implementation from the software stack and the end application. Huge accuracy improvements can be obtained for speech recognition by using higher N-gram language models, especially if they can be made application specific. However, building a hardware architecture that works equally well with any N-gram model is a herculean task. This made us move to the multi-pass decode approach, where we can use a simpler language model in the first pass. We chose to move this first pass to hardware. However, we observed that using a simpler

model reduced the recognition accuracy, even if the language model in the second decode pass was a sophisticated trigram model. This was because the output of the first pass did not contain the best hypothesis for the second pass to find it. So we tried increasing the beam width for the first pass, so as to include the best hypothesis. However, we realized that expanding the beam width was not going to help much as the unigram/bigram model did not have enough knowledge to choose the best path at any node in the search. This made move to the N-best approach which keeps track of N-best paths at each node. We found the complexity of this exact N-best algorithm[10] to be impractical and hence moved to approximate N-best approaches[11]. Our literature review lead us to the word dependent N-best algorithm which is well suited for use with a unigram/bigram language model. We decided to implement this algorithm for the first pass of decode in hardware and used a trigram language model with a 1-best Viterbi search for the second pass. After this, we explored the design space for multiple acoustic models and word vocabularies to find an optimal design point. The proposed hardware architecture is design to work at this design point of 8000 senones, 8 gaussians and a word vocabulary of 64K words, while achieving real time accuracy. On the way to achieving this, we came up with some innovative ideas which improve performance, power consumption and overall efficiency of the system.

7.2 Future Work

In this work, we have tried to explore the relationship between the complexity of hardware design and the end accuracy of speech recognition. We have enabled parallelism of multiple stages of the speech recognition algorithm. Since the observation probability calculation stage and decoding stage are now in hardware, the next extension would be

to look at any bottlenecks in the front end stage. The front end employed by sphinx is very simple. This stage can easily get complicated and computationally intensive if we want to combat noise, echo, reverberation and other ambient conditions. At this point, it will be worth investigating if a hardware implementation would be useful for these tasks.

We implemented the word dependent N-best algorithm in this work. At this time, there are no other hardware implementations which perform N-best search. We believe that N-best search is inherently suited for hardware implementation and memory access optimization since it provides more sequentially and locality in memory accesses. It would be interesting to see how other N-best approaches perform in hardware.

In this work, we divided the decode stage into two passes, but it is not clear if two is the optimal number of decode stages. Having more than two stages of decode may end up providing better performance benefits.

The A* stack decoding algorithm is a single pass algorithm which can efficiently perform and N-best search. Since it uses stacks to store paths in the search, it might be able to provide optimized memory accesses. If this can be efficiently implemented in hardware, it has the potential to outperform other multipass N-best algorithms.

REFERENCES

[1] B. Matthew, A. Davis, and Z. Fang, “A low-power accelerator for the sphinx 3 speech recognition system,” *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2003.

[2] D. Chandra, “Speech recognition co-processor,” Ph.D. dissertation, NC State University, 2007.

[3] U. Pazhayaveetil, “Hardware implementation of a low power speech recognition system,” Ph.D. dissertation, NC State University, 2007.

[4] Y. kyu Choi, “Vlsi for 5000-word continuous speech recogntion,” *International Conference for Acoustics, Speech and Signal Processing*, 2009.

[5] Y. kyu Choi and K. You, “A real-time fpga-based 20 000-word speech,” *IEEE Transactions on Circuits and Systems - I*, 2010.

[6] D. Chandra, U. Pazhayaveetil, and P. Franzon, “Architecture for low power large vocabulary speech recognition,” in *SOC Conference, 2006 IEEE International*, sept. 2006, pp. 25 –28.

[7] M. Hwang and Mei-yuh, “Subphonetic modeling with markov states - senone,” *International Conference for Acoustics, Speech and Signal Processing*, 1992.

[8] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, “Class-based n-gram models of natural language,” *Comput. Linguist.*, vol. 18, no. 4, pp. 467–479, Dec. 1992. [Online]. Available: http://dl.acm.org/citation.cfm?id=176313.176316

[9] A. Viterbi, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,” *Information Theory, IEEE Transactions on*, vol. 13, no. 2, pp. 260 –269, april 1967.

[10] R. Schwartz and Y.-L. Chow, “The n-best algorithms: an efficient and exact procedure for finding the n most likely sentence hypotheses,” in *Acoustics, Speech, and Signal Processing, 1990. ICASSP-90., 1990 International Conference on*, apr 1990, pp. 81 –84 vol.1.

[11] R. Schwartz and S. Austin, “A comparison of several approximate algorithms for finding multiple (n-best) sentence hypotheses,” in *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, apr 1991, pp. 701 –704 vol. 1.

[12] D. Jurafsky and J. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, ser. Prentice Hall Series in Artificial Intelligence. Pearson Prentice Hall, 2009, pp. 285 –359. [Online]. Available: http://books.google.com/books?id=fZmj5UNK8AQC

[13] H. Van Hamme and F. Van Aelten, "An adaptive-beam pruning technique for continuous speech recognition," in *Spoken Language, 1996. ICSLP 96. Proceedings., Fourth International Conference on*, vol. 4, oct 1996, pp. 2083 –2086 vol.4.

[14] CMU, *"Sphinx 3.0"; http://www.cmusphinx.sourceforge.net/*. [Online]. Available: http://www.cmusphinx.sourceforge.net/

[15] L. R. Rabiner, "Tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, 1989.

[16] S. Khan, G. Sharma, and P. Rao, "Speech recognition using neural networks," in *Industrial Technology 2000. Proceedings of IEEE International Conference on*, vol. 1, jan. 2000, pp. 432 – 437 vol.2.

[17] S. Katz, "Estimation of probabilities from sparse data for the language model component of a speech recognizer," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 35, no. 3, pp. 400 – 401, mar 1987.

[18] M. Mohri, "Finite-state transducers in language and speech processing," *Comput. Linguist.*, vol. 23, no. 2, pp. 269–311, Jun. 1997. [Online]. Available: http://dl.acm.org/citation.cfm?id=972695.972698

[19] H. Ney, R. Haeb-Umbach, B.-H. Tran, and M. Oerder, "Improvements in beam search for 10000-word continuous speech recognition," in *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, vol. 1, mar 1992, pp. 9 –12 vol.1.

[20] U. Pazhayaveetil, D. Chandra, and P. Franzon, "Flexible low power probability density estimation unit for speech recognition," in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, may 2007, pp. 1117 –1120.

[21] F. L. Vargas, R. Dutra, and R. Fagundes, "An fpga based viterbi algorithm implementation for speech recognition," *Acoustic, Speech and Signal Processing*, 2001.

[22] S. Yoshizawa and N. Hayasaka, "Scalable architecture for word hmm-based speech recognition and vlsi implementation in complete system," *IEEE Transactions on Circuits and Systems - I*, 2006.

[23] J. Bourke Patrick and A. Rutenbar Rob, "A low-power hardware search architecture for speech recognition," in *Speech Communication , 2008. INTERSPEECH 08. Proceedings., Nineth International Conference on*, 2008.

[24] M. Li and T. Wen, "Hardware software co-design for viterbi decoder," *International Conference on Electronic Packaging Technology and High Density Packaging*, 2008.

[25] O. Cheng, W. Abdulla, and Z. Salcic, "Hardware software codesign of automatic speech recognition system for embedded real-time applications," *Industrial Electronics, IEEE Transactions on*, vol. 58, no. 3, pp. 850 –859, march 2011.

[26] P. Li and H. Tang, "Design of a low-power coprocessor for mid-size vocabulary speech recognition systems," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 58, no. 5, pp. 961 –970, may 2011.

[27] J. Bourke Patrick and A. Rutenbar Rob, "A high-performance hardware speech recognition system for mobile applications," in *SRC Techcon, 2005, Proceedings., Nineth International Conference on*, 2008.

[28] K. Vertanen. (2006) Cmu sphinx acoustic models for us english. [Online]. Available: http://www.keithv.com/software/sphinx/us/

[29] ——. (2006) Csr lm-1 language model training recipe. [Online]. Available: http://www.keithv.com/software/csr/

[30] C. Sanderson and K. K. Paliwal, "Effect of different sampling rates and feature vector sizes on speech recognition," *Speech and Image Technologies for Computing and Telecommunications 161*, 1997.

[31] H. Hirsch, K. Hellwig, and S. Dobler, "Speech recognition at multiple sampling rates," *Eurospeech*, 2001.

[32] W. Zhang, L. He, Y.-L. Chow, R. Yang, and Y. Su, "The study on distributed speech recognition system," in *Acoustics, Speech, and Signal Processing, 2000. ICASSP '00. Proceedings. 2000 IEEE International Conference on*, vol. 3, 2000, pp. 1431 –1434 vol.3.

[33] G. M. S. S. Thorsten Grtker, Stan Liao, *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[34] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '03. New York, NY, USA: ACM, 2003, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/944645.944651

[35] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, jan.-june 2011.

[36] N. Dhanwada, I.-C. Lin, and V. Narayanan, "A power estimation methodology for systemc transaction level models," in *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '05. New York, NY, USA: ACM, 2005, pp. 142–147. [Online]. Available: http://doi.acm.org.prox.lib.ncsu.edu/10.1145/1084834.1084874

[37] Micron, "Calculating memory system power for ddr sdram," Tech. Rep., 2005. [Online]. Available: http://www.ece.umd.edu/class/enee759h.S2005/references/dl201.pdf

[38] O. Bapat, "Design of ddr2 interface for tezzaron tsc8200a octopus memory intended for chip stacking applications," Master's thesis, NC State University, 2010.

[39] U. of California at Berkeley. (2009, Nov.) Nist speech recognition scoring toolkit. [Online]. Available: http://www1.icsi.berkeley.edu/Speech/docs/sctk-1.2/sctk.htm

[40] K. Vertanen, "Baseline wsj acoustic models for htk and sphinx: Training recipies and recognition experiments," *Technical Report, Cavendish Laboratory*, 2006.