A 1000-Word Vocabulary, Speaker-Independent, Continuous Live-Mode Speech Recognizer Implemented in a Single FPGA

Edward C. Lin, Kai Yu, Rob A. Rutenbar, Tsuhan Chen
Carnegie Mellon University
Pittsburgh, PA 15213 U.S.A.
{eclin, kaiy, rutenbar, tsuhan}@ece.cmu.edu

ABSTRACT
The Carnegie Mellon *In Silico Vox* project seeks to move best-quality speech recognition technology from its current software-only form into a range of efficient all-hardware implementations. The central thesis is that, like graphics chips, the application is simply too performance hungry, and too power sensitive, to stay as a large software application. As a first step in this direction, we describe the design and implementation of a fully functional speech-to-text recognizer on a single Xilinx XUP platform. The design recognizes a 1000 word vocabulary, is speaker-independent, recognizes continuous (connected) speech, and is a "live mode" engine, wherein recognition can start as soon as speech input appears. To the best of our knowledge, this is the most complex recognizer architecture ever fully committed to a hardware-only form. The implementation is extraordinarily small, and achieves the same accuracy as state-of-the-art software recognizers, while running at a fraction of the clock speed.

Categories and Subject Descriptors
C.3 [**Special-Purpose and Application-Based Systems**]: Signal processing

General Terms
Algorithms, Performance, Design

Keywords
Speech Recognition, FPGA, DSP, In Silico Vox

1. INTRODUCTION
Speech is an exceptionally attractive modality for human-computer interaction: it is "hands free"; it requires only modest hardware for acquisition (a high-quality microphone or microphones); and it arrives at a very modest bit rate. Recognizing human speech, especially continuous (connected) speech, without burdensome training (speaker-independent), for a vocabulary of sufficient complexity (60,000 words) is very hard.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'07, February 18–20, 2007, Monterey, California, USA.
Copyright 2007 ACM 978-1-59593-600-4/07/0002...$5.00.

There has been significant progress over the last few decades on this important problem [1]. However, whether supporting dictation on a desktop computer, or customer telephone input for a commercial call center, or recognizing numbers and contacts in a cell phone, all high-quality recognizers exist today as *software* running on some CPU. We believe this is extraordinarily limiting to ultimate performance, vastly inefficient in its use of silicon, and in need of a paradigm shift. Like graphics chips over the last decade, we believe that the application is simply too important, too performance hungry, and too power sensitive, to stay as a large software application.

Audio mining is one such performance-hungry application worth mention. As video content increasingly goes on-line, there is a growing need to search it. One natural indexing method is the accompanying audio track. However, today's indexing requires text-based search of closed-captioning for these video sources. What if we could, instead, directly search the voice data in video streams? What if we could search at 10X, 100X, or 1000X faster than real-time (i.e., faster than the rate at which the speech was produced)? We do not believe tweaks to software versions of today's state-of-the-art recognizers, or the near-term roadmap for processor improvements, will yield the multiple orders-of-magnitude speedups we seek. At Carnegie Mellon, the *In Silico Vox* project [2] is working to design and deploy a range of hardware-only implementations of recognizers operating at these performance levels.

Of course, a hardware-based recognizer is hardly a new idea. However, prior efforts suffer from a range of serious flaws. There are many small-vocabulary architectures, e.g. dynamic time-warp strategies [3][4] that cannot scale to the large-vocabulary case we target. Such approaches were abandoned by the speech community at least a decade ago. Critical bottleneck steps in the algorithms have been migrated to hardware coprocessor forms [5], but suffer from serious Amdahl's Law limitations to overall improvement. Still other studies have proposed interesting multi-processor architectures, but these either fail to demonstrate more than cursory performance improvement [6], or fail to demonstrate performance on other than trivial benchmarks [7]. A notable exception is the complete VLSI-based recognizer built at Berkeley in the early 1990s [8]. However, by the standards of a 2007 recognizer, its core recognition strategy is quite primitive.

As a first step toward a fully hardware-based recognizer, we explore the problem of moving a modern recognition architecture onto FPGA hardware. We describe in detail the architecture, design and implementation of a complete speech recognizer in a single FPGA. Our principal objective is to build a fully functional

recognizer entirely in custom hardware, and use this implementation to understand the essential tradeoffs inherent in a hardware-only solution. As a consequence, we choose not to exploit any on-chip processor cores available on our FPGA. The problem of how best to deploy custom logic alongside embedded processor resources for speech recognition is an interesting one—but it is not the problem we address in this paper. Our overriding interest is to use the FPGA prototype to gain insight into the complexity of a hardware solution; we believe our ultimate platform will be a custom silicon recognizer.

We use the Xilinx XUP development board as our implementation platform [9]; this has the twin advantages of having integrated DRAM in sufficient quantity, and on-board audio input. However, moving a modern recognizer onto the XUP system proves to be a significant design challenge: our implementation must fit in a footprint with substantially fewer than 1M equivalent logic gates (i.e., roughly 1mm^2 of silicon in a modern CMOS ASIC process) and less than 1MB of on-chip SRAM. Achieving a functional recognizer creates a challenging set of size/performance tradeoffs. We target a 1000-word command/control benchmark called *Resource Management* [10]. This is small by the standards of a dictation-class recognizer, but has three desirable features: (1) it is widely used in the speech recognition community; (2) it is vastly larger than most recent hardware studies [7]; (3) it is the largest benchmark that fits on the rather small XUP platform, but can still run live. An earlier version of this work, with emphasis on its impact on the speech recognition community, appears in [11]. The XUP result was first mentioned in [12], but without details of the FPGA-based architecture and implementation. In this paper, we offer those details.

The paper is organized as follows. Section 2 reviews the theory and algorithms of speech recognition, as embodied in the CMU *Sphinx 3.0* [13] software recognizer, which we adopt as our reference model. Section 3 then profiles the performance of the Sphinx 3.0 reference software. Section 4 describes the components of our overall recognizer architecture. Section 5 describes the XUP-based FPGA prototype. Section 6 describes experimental results: a fully functioning recognizer, running at 50MHz, capable of recognition at roughly 2.3X slower than real-time. Finally, Section 7 offers concluding remarks.

2. ABOUT SPEECH RECOGNITION

Speech recognition is the conversion of spoken words to text. As the first step toward rendering a state-of-the-art recognizer in hardware form, we start with a software implementation as a reference model, and dissect it. That reference platform for us is the Sphinx 3.0 recognizer, a speaker-independent speech recognition system which can handle *continuous* speech (no pauses between words). The Sphinx 3.0 architecture (and indeed, today the most common recognizer architecture) is depicted in Figure 1.

First, analog speech input is sampled and digitized, then divided into 10 ms blocks called *frames*. Next, in the *acoustic frontend* stage, signal processing algorithms are applied to each frame to generate a *feature vector*, which contains all the acoustic information for the frame. After that, *Gaussian mixture model (GMM) scoring* is performed to calculate the probabilities of all the possible sounds that could have been pronounced. Finally,

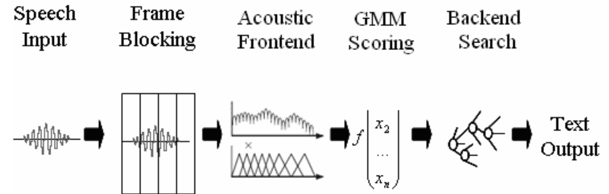

Figure 1. Speech recognition decoding flow

these values are used by the *backend search* to find the most probable word sequence. The following subsections will describe each individual stage in more detail.

2.1 Acoustic Model

Speech recognition is built upon a layered model [1]. In the most fundamental layer, acoustic information is used as the building block for piecing together speech. In the English language, this would be the approximately 50 differentiable sounds called *phonemes*. When these phonemes are pronounced, the acoustic realization, or *phone*, varies depending on the phone voiced directly before and after it. To accurately depict this effect, speech recognizers use *triphones*, phones in the context of the preceding and succeeding phones, to build words. Triphones are further divided into smaller sub-acoustic units, which are the really the first and lowest layer of the speech recognition hierarchy. Our implementation divides each triphone into three sequential acoustic states, i.e, three *atomic sounds* that we must try to recognize. Looking across all the data heard across all speech training inputs, we cluster these into a smaller, more manageable set of atomic sounds called *senones*. Thus, senones combine to form triphones, and triphones combine to form words.

2.2 Hidden Markov Models

Hidden Markov Models (*HMMs*) [14] are the most common method to represent speech at the acoustic level described above, i.e., from microphone input, to senones, to triphones. This is in part due to their ability to discriminate among a set of connected sounds by using a probabilistic approach. HMMs are composed of *states* and *observations*. States, like in Markov models, contain transition probabilities to other states that only depend on the current state. However, the state sequence is unknown, and only through a series of observations can the most likely hidden sequence of states be computed. To achieve this, each state must also contain the probability of each observation (each atomic sound) when within that state.

The following example will better illustrate the difference between HMMs and Markov models. Consider a simple weather system where it is either *sunny* or *raining*, and the probability of carrying an umbrella in the two climates is already known. In a standard Markov model, this system would be represented by two states, sunny and raining, with transition probabilities between the two states. Now consider a person who is interested in the weather, but cannot directly observe the climate. Instead, this person can only observe whether a daily visitor is carrying an umbrella. This situation is best represented with a *hidden* Markov model, or HMM, where the system looks like the Markov model but the sunny and rainy states now carry the additional information as to the probability of carrying an umbrella in that

climate. Now, based on a series of daily observations of whether this visitor is carrying an umbrella or not, the most likely sequence of daily climates can be computed.

But HMMs are just the representation of the connected sounds. To determine the most likely state sequence, i.e., to determine which set of heard atomic sounds is most likely what was uttered by the speaker, the *Viterbi* search algorithm is used [15]. It is time-synchronous, and for every time interval (10ms frame), it steps through all the possible transitions. For each state, the highest probability of reaching that state at the end of the time interval is stored. To calculate the maximum probability of reaching state j at time t:

$$P_t(j) = \max[P_{t-1}(i)a_{ij}]b_j(O_t) \text{ for all states that transition to } j \quad (1)$$

where a_{ij} is the transition probability from state i to state j, b_j is the observation probabilities of state j, and O_t is the observation at time step t. To trace the most likely state sequence, each state needs to additionally store the predecessor state that led to the highest probability.

Although an exhaustive search guarantees the Viterbi algorithm will find the most likely state sequence, it also causes the search space to grow exponentially with each time step. To make the search more manageable and complete within a reasonable amount of time, at each time step the sequences with probabilities below a certain threshold are pruned away. This algorithm is called the *Viterbi beam search*, and the unpruned states are referred to as *active*. Although this algorithm is now sub-optimal and may prune away the most likely state sequence, the resulting most likely state sequence may still be the same as the most likely state sequence.

In our speech recognition implementation, 4-state HMMs are used to represent triphones. The first three states represent senones (atomic sounds we have heard during off-line training over a large number of different speakers, i.e., the library of lowest level acoustic units we can start matching/recognizing). The fourth and final state is a *null* state used to tie the HMMs together into words. Each word has its own HMM sequence, and the observation (senone) probabilities are calculated by the GMM scoring.

2.3 Language Model

One issue that the acoustic model cannot handle by itself is *word ambiguity*. Word ambiguity can occur in several forms such as homophones and word boundaries. With homophones, words are indistinguishable to the ear, but are different in spelling and meaning. The words “to”, “two”, and “too” are such examples. In continuous speech, word boundaries can also be challenging. For instance, the word “conundrum” can be misconstrued as “con nun drum” if not correctly parsed. The use of a *language model* resolves these issues by considering phrases and words that are more likely to be uttered. When a transition occurs at the final state of the last HMM of a word, the language model is used to analyze possible *following* word candidates given the *previous* word(s) that were believed to have been recognized. Likely word candidates have their probabilities boosted. Because the memory storage required grows exponentially with the number of previous words taken into account, only single word, word pairs, and/or word triples, commonly referred to as *unigrams, bigrams* and *trigrams*, are considered in the language model.

2.4 Acoustic Frontend

The acoustic frontend takes each frame (each 10ms sample of live speech) and distills the relevant acoustic information through a series of signal processing operations. While there are several methods to extract the acoustic data, the most popular form, *mel frequency cepstral coefficients* (mfcc), will be explained here. Its operations are based on the physiology of the human ear and relatively straightforward to implement. To generate the mfcc values per frame, a pre-emphasis filter is first used to boost the energy in the high frequencies which contain important acoustic information. A Hamming window and 512-point FFT are then applied, and the power content of the frequencies, or spectrum, is computed. A triangular filter bank designed to extract mel frequencies is applied to simulate the spectral resolution of the ear, and an inverse discrete cosine transform converts the data back to the time domain and into the mfcc values.

To generate the acoustic feature vector, the mfcc values undergo cepstral mean normalization, which average the mfcc values over a certain time period. This helps reduce the effect of background noise on the mfcc values, and the time period varies from the entire speech sample (batch mode) to a subset of the preceding frames (live mode). Since human hearing is also related to the change velocity and acceleration of the mfcc values, the first and second derivative of the normalized mfcc values are taken to generate the acoustic feature vector.

2.5 GMM Scoring

GMM scoring assigns probabilities to each of the senones per frame based on the acoustic feature vector. To achieve speaker-independent decoding, variations in senone pronunciation (i.e., for different voices, accents, genders, etc.) must be accommodated, so senones are not represented by a single point but rather a weighted sum of Gaussian probability density functions called a *Gaussian mixture model*. To calculate the individual senone probabilities, this equation is computed once per frame per senone:

$$p = \sum_{j=1}^{n} \frac{w_j}{\sqrt{2\pi\Lambda_j^2}} e^{\sum_{k=1}^{t} \frac{(x_k - \mu_{j,k})^2}{2\sigma_{j,k}^2}} \quad (2)$$

where n is the number of Gaussian mixtures per model, t is the size of the acoustic feature vector, w_j is the weight, Λ_j^2 is the generalized variance, x_k is an element from the feature vector, $\mu_{j,k}$ the mean of the Gaussian pdf, and $\sigma_{j,k}^2$ the variance. Besides the feature vector, all of the other values are pre-computed constants derived from the training data. To prevent floating point underflow problems when the senone probabilities are multiplied together to find the most likely sequence of sounds, the logarithms of the senone probabilities are calculated.

2.6 Backend Search

The backend search takes the senone probabilities per frame, and finds the most likely sequence of words. To do this, it applies the Viterbi algorithm to all the active states within an active HMM of all the active word candidates. A particular set of complications here is the fact that backend search is not the process of scoring transitions within *one* single HMM. In addition to handling transitions between atomic sounds (senones) in one triphone, we

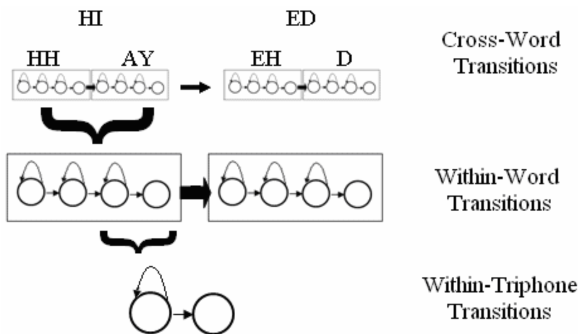

Figure 2. Three types of transitions in HMMs

must also handle transitions from one triphone to another, inside one word (*within-word* transitions), and transitions from the ending triphone of one word, to the beginning triphone of the follow-on word (*cross-word* transitions). The three types of transitions are illustrated in Figure 2. For the *within-triphone* and *within-word* transitions, the possible transitions are limited by the HMM structure and word pronunciation. For the *cross-word* transitions, any transition between words is possible, so the language model is applied to help recognition accuracy.

3. PROFILING THE SPHINX 3.0 REFERENCE ARCHITECTURE

Given the brief overview of the major components of a state-of-the-art recognizer, the next obvious question is: *Why is this computationally so difficult?* To answer this, we profiled Sphinx 3.0 to determine where it spends its time, and what potential gains our FPGA implementation could exploit. While many speech recognizers, including earlier versions of Sphinx, have been profiled before [16][17], Sphinx 3.0 has significant differences that lead to its high accuracy. For these experiments we used a 1000 word vocabulary *Resource Management* (RM) task composed of military command and control phrases. The speech model contained 8 Gaussians per GMM, a 39-dimensional feature vector, 1000 unigrams, 2385 bigrams and no trigrams. On a 2.8 GHz Intel Xeon workstation with 1 GB of RAM, Sphinx 3.0 was able to decode 3.7 times faster than real-time. As seen in Figure 3, it spent 76% of the time on the GMM scoring stage, 24% of the time on backend search, and a negligible amount of time on the acoustic frontend. This is consistent with published results, though GMM scoring made up an abnormally large amount of execution time due to the small language model used.

To find what factors limit decoding speed, we used Simplescalar [18] and modified parameters to see their effect on cycles simulated. While the DL1 miss rate is small (5.40% for a 16 KB cache with 16 byte blocks), this is due to the small size of the language model. As the language model size increases, so does the miss rate. For example, the miss rate increases to 24.41% for the 60,000 word vocabulary *Broadcast News* task [10]. We found that accessing data from memory had the largest effect, and if we had perfect memory, where DRAM and caches all had single cycle access, the cycles decreased by 36%. We also modified the parameters to better match that of a custom hardware design to get an estimate of how much improvement our design would achieve. By reducing cache accesses to a single cycle, and increasing the number of functional units and the number of instructions fetched/decoded/issued, the cycles decreased by 34%.

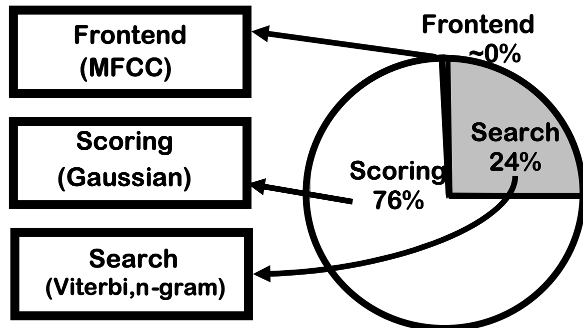

Figure 3. Timing breakdown of Sphinx 3.0 using 1000 Word Resource Management Task

Our results show that a custom hardware speech recognition can achieve significant gains compared to a software one. It also shows that since the acoustic frontend takes so little time, it should be designed to minimize area, while the GMM scoring and backend search should be designed to maximize decoding speed. Finally, if we plan to extend this design to larger language models, when designing the architecture we need to be aware that there is little memory locality.

4. HARDWARE SPEECH RECOGNITION

For our FPGA-based speech recognition system, we settled on a live-mode design running the 1000-word RM task. The target platform was the Xilinx XUP development board with a Xilinx Virtex-IIPro XC2VP30. The goal for our hardware recognizer is to achieve a *modest* decoding speed (the XUP memory subsystem is not fast enough for real-time), while using the *fewest* hardware resources and running at the *slowest* possible clock speed. In this section we discuss our approach to developing our architecture, the datapath of our design, and the optimizations we made to our hardware speech recognizer to increase performance. We use *decoding speed* and *word error rate* (WER) as metrics to evaluate speech recognition performance.

4.1 Cycle-Accurate Hardware Simulator

Because going directly to the FPGA is difficult and time consuming, we followed a conventional system simulation methodology before placing our design on the FPGA. To begin, we developed a hardware simulator in C++ to prototype different hardware designs and to converge on an optimal architecture. For success, we needed to be able to guarantee that the simulator was both cycle-accurate and bit-true. All internal state elements of the hardware after every cycle must match the exact data value when compared with a software reference model. Our simulator maintains these characteristics. Of course, with software speech recognition already a CPU-intensive task, simulating a hardware recognizer at bit-level requires significantly more processing power. It was common for several seconds of speech to require several CPU *days* to simulate.

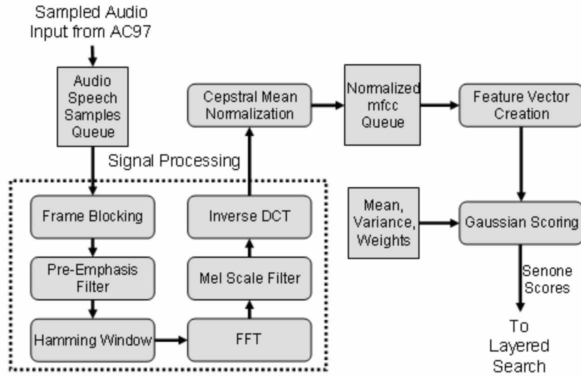

Figure 4. Acoustic Frontend and GMM Scoring datapath

4.2 Architecture

After several revisions of our simulator, we established the datapath to be implemented on the FPGA. The design has the following attributes.

4.2.1 *Acoustic Frontend*

As determined from the profiling, decoding speed is not an issue in this stage, so the focus was on minimizing resource usage. Since the algorithm is sequential, the design attempts as much resource sharing as possible. For example, the memories and multipliers used for the FFT are reused in the later parts of decoding. To save space, many of the constants for small individual data structures were combined into a single dual-ported block RAM (BRAM).

4.2.2 *GMM Scoring*

The GMM scoring algorithm is conceptually simple and straightforward to implement. But because it is applied per senone per frame, this stage turns out to consume a significant amount of decoding time. GMM scoring is limited by the time it takes to fetch the GMM constants from DRAM. If the GMMs are computed per frame sequentially, the read memory bandwidth required would be 143 MB/s. However, this is the worst case and can be reduced by replicating the GMM scoring unit many times. Each instance can compute senone scores independently of each other while sharing the same constants read from main memory. This way, if the GMM scoring unit is replicated N times, the bandwidth required would also decrease by N times. For our

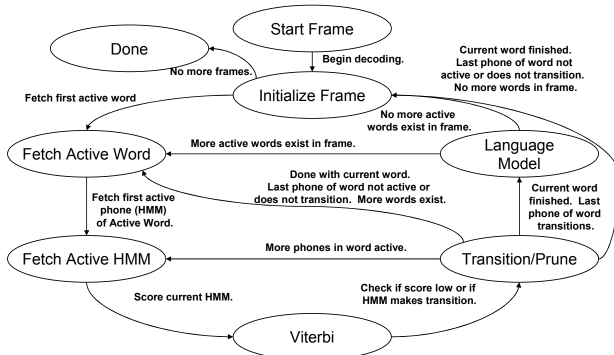

Figure 5. Backend Search state machine

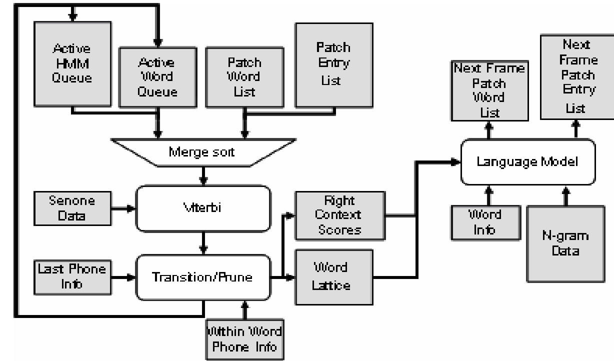

Figure 6. Backend Search datapath

design, while it would have been beneficial to replicate the GMM scoring unit, due to area constraints we were unable to. The final architecture for the acoustic frontend and GMM scoring is depicted in Figure 4.

4.2.3 *Backend Search*

The backend search data flow can be described as a finite state machine as shown in Figure 5. Each state in the diagram corresponds to a part of the hardware datapath shown in Figure 6. We briefly describe each state and its corresponding role and requirements in the backend search datapath.

- The *Start Frame* and *Done* states determine the beginning and end of decoding. The Initialization state sets up the initial values of data for each frame.
- The *Fetch Active Word* state retrieves word information for a single active word stored in the *Active Word Queue* or the newly entered word list from the cross-word transitions, called the *Patch List*. The words are decoded alphabetically.
- Once a word is selected, active word HMMs are either *fetched* from the Active HMM Queue or a new HMM is *activated* when a new word is entered by the Patch List in the Fetch Active HMM state.
- Next, each active HMM goes through the Viterbi scoring state. While the Viterbi computation itself is not computationally demanding, it does require several memory lookups.

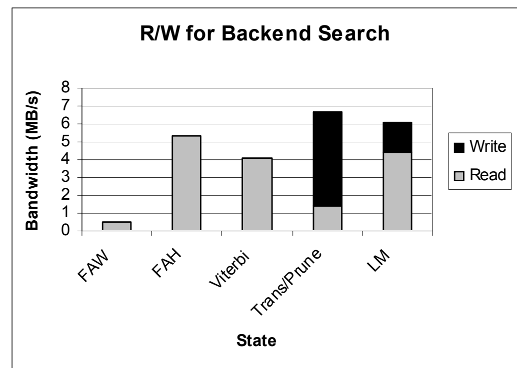

Figure 7. Backend Search bandwidth per state

- In the *Transition/Prune* state, an active HMM can make a transition to the next phone, get pruned, and/or get written back for the next frame.
- If an HMM makes a word transition, the *Language Model* is used to determine possible words that can get generated. If a word is deemed likely, it gets stored into the Patch list for the next frame. Much like Viterbi, this state requires many memory lookups.

The overall backend search requires ~23.3 MB/s of I/O, broken down into ~15.4 MB/s of reads and ~7.9MB/s of writes. A breakdown of I/O by states is shown in Figure 7. It is interesting to note that the fetching and storing back of active HMMs and accessing the language model dominate memory accesses.

4.3 Architectural Modifications

In this section we highlight different functional modifications which set our hardware design apart from Sphinx 3.0, and introduce structural modifications which use efficient hardware methods to increase performance.

4.3.1 Functional Modifications

Custom Bit-widths: In the acoustic frontend and GMM scoring stages of Sphinx 3.0, numbers are represented as 32-bit floats. Through experimentation we found that these numbers do not require such precision to achieve the same WER. By using custom bit-widths, we reduce the required memory storage and chip area of our design without sacrificing WER. On average the modification reduces bit-widths by 33%, and decreases memory required to store GMM constants by 50%.

Log Lookup Table: In the GMM scoring stage, the log probability of each senone needs to be calculated. In software, this is accomplished when the log probability of each GMM mixture is computed, and then summed with the help of a ~100,000 element lookup table. In hardware, storing this large lookup table on-chip would be very costly. Thus, in its place we use a more complex interpolation using four third-order polynomials. This equates to replacing a large memory lookup table with extra logic gates which proves to be more area efficient.

Pruning Threshold: Another major functional change we make is how the pruning threshold per frame is determined. In Sphinx 3.0, each frame begins by updating the state probabilities of *all* the active HMMs. Then based on the highest probability state, the pruning threshold is established. Next, they go through all the active HMMs again to find active HMMs whose probabilities fall below the threshold are pruned away. This approach requires going through the active HMMs *twice*, which practically doubles the memory accesses and hinders performance proportionally. This algorithm also forces the transition computations to occur *after* all the HMMs are updated, which reduces the effective parallelism. We avoid these constraints by using the best score from the *previous* frame to determine the pruning threshold. By setting the threshold to be a function of the previous frame, active HMMs can immediately be pruned or transitioned after updating all its state probabilities, reducing memory bandwidth. The modification also allows for different active HMMs to be doing computation in different stages at the same time. This method can potentially decrease the number of active HMMs per frame, but through simulation we show there is a negligible increase in WER from 10.88% to 10.95%.

4.3.2 Structural Modifications

Active HMM Storage: In Sphinx 3.0's search stage, an HMM requires 40 to 52 bytes of storage. We compressed this to 28 to 36 bytes without affecting functionality. We also modify how the active HMMs are stored in memory. Instead of as linked data structures in Sphinx, we store active HMMs consecutively in memory as a queue. By replacing a random access memory structure with a predictable one, we can retrieve data from memory more effectively.

Cross-word Transitions: Normally when a cross-word transition occurs it requires retrieving the first HMM of the word being transitioned into, and possibly updating the probability of the first state of the HMM according to the Viterbi algorithm. With a large word candidate list which a given word can transition to, this can be a very memory intensive operation. To eliminate this bottleneck, we store *all* cross-word probabilities in an on-chip memory called the Patch List. This approach filters off-chip memory accesses, and quickly handles probability updates. As stated in the previous subsection, the Patch List updates the Active HMM list in the next frame.

Pipelining: We use pipelining throughout our design to increase throughput and allow for a fast clock frequency. One such example is with the GMM mixture computation, where Gaussian probabilities in 6 different dimensions are computed at the same time.

Scheduling: In our design, two stages require use of the DRAM, GMM scoring and backend search. We use a token passing scheduler to handle priority of requests between the stages. Within the GMM stage only a single unit requires the DRAM. In the backend search stage there are many sub units which require the DRAM. A fixed priority scheduler is used to determine which sub unit gets access.

5. FPGA-BASED SPEECH RECOGNIZER

For our live-mode hardware speech recognition system to be fully functional on the Xilinx XUP development board, several key system issues also needed to be addressed, including DRAM initialization, DRAM bandwidth, decoding speed, and I/O. These topics are discussed in this section.

5.1 DRAM

Before decoding can begin, the recognizer requires an initialization period to move static data (all the scoring information for all the GMMs, HMMs, and language models) from a Compact Flash card to DRAM. This is accomplished by using one of the on-chip PowerPC processors. Once the data is transmitted, control is then passed to the speech recognition core where decoding may begin. During decoding, the speech recognition core directly accesses the DRAM through the Processor Local Bus (PLB). We selected this method of accessing DRAM for practicality since Xilinx provides its own PLB-DDR controller in its library. A higher bandwidth solution would be to design our own memory controller to DRAM, which given time constraints, we decided against.

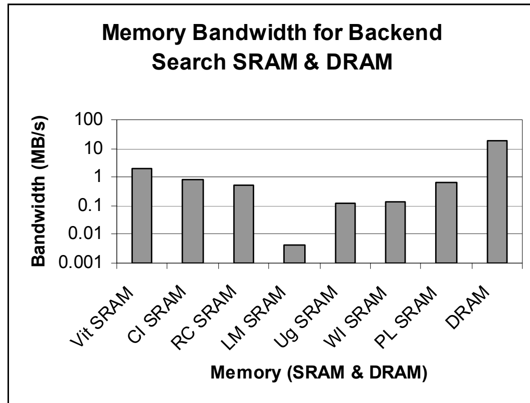

Figure 8. Backend Search SRAM and DRAM bandwidth

A major bottleneck in decoding is DRAM memory bandwidth through the PLB. With a maximum burst length of 16 and a data bus size of 64 bits, the maximum bandwidth allowed to DRAM through the PLB is ~200MB/s. Bandwidth is limited by bus communication and the 100MHz PLB speed. In our design, only reads are able to be bursted. Read transfers fully utilize the 64-bit data bus, while writes are masked, and transferred at 32-bits. Each request is handled one at a time; there are no split transactions. In order to speed up our design we use custom burst lengths depending on requestor's access patterns. Only DRAM command transfers are handled by the PLB so there is no contention for the bus by other devices.

Because of these bandwidth and also area constraints, we chose to run our design at a modest 50MHz. By doing so, synchronization is needed to handle DRAM transfers. A dual-ported BRAM is used to buffer read bursts and handshaking is used to coordinate signals between the two clock domains.

5.2 Buffering for Live Mode Decoding

If the decoding speed of a frame is slower than the frame length, frames will be dropped unless some buffering mechanism is introduced. While the acoustic frontend and GMM scoring stages have deterministic decoding times shorter than the frame length, the backend search decoding time varies depending on the number of active HMMs which are alive in each frame. Therefore, to prevent dropping frames, we introduce a buffer right after normalizing the mfcc values. This way the essential information is captured by a mere 13 values, which is much more efficient than a buffer placed elsewhere.

5.3 System-Level Environment

The Xilinx XUP development board has the capability to support a microphone, VGA monitor, and push buttons. We use each of these devices to aid our recognizer in the following manner. When a push button is pressed, the speech from the microphone is sampled by the AC97 audio codec, and fed to the speech recognition core on the FPGA, where decoding immediately begins. Another push button press signifies the end of recording. Once the recognizer is done decoding, the decoded word hypothesis is displayed on the VGA.

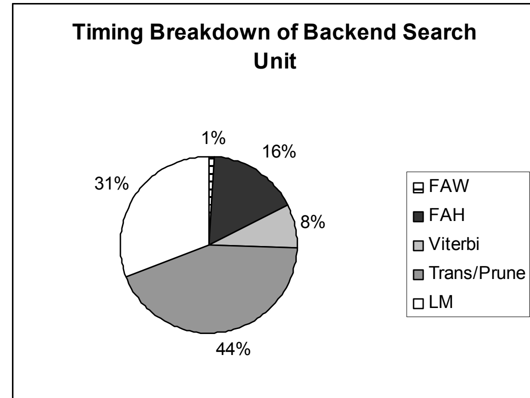

Figure 9. Timing breakdown of the Backend Search stage

6. EXPERIMENTAL RESULTS

In this section we discuss our initial simulation results, verify functionality, and discuss the results of our design on the Xilinx XUP development board.

6.1 Hardware Simulator Results

To estimate the decoding speed of the proposed architecture, we modeled the design running at 50 MHz on our cycle-accurate hardware simulator and created a memory model which simulated 64-bit DRAM accesses through the PLB as in the FPGA. The model handles burst and non-burst memory requests. Each SRAM access is assumed to be a single cycle.

A critical bottleneck is how memory should be divided in the backend search stage. After profiling the bandwidth and size of each memory structure and looking at access patterns, we determined a good partition for which data should be placed in SRAM or DRAM (Figure 8).

From our final simulator results we estimate that this hardware design can decode at roughly ~2.2 slower than real-time or ~0.5 time faster than real-time; cycle breakdown by stage is as follows: ~0% for acoustic frontend, 37% for GMM scoring, and 63% for backend search. The timing breakdown of the backend search stage can be seen in Figure 9. The reason why the backend search takes so much time for decoding is because of DRAM latency. 52% of the backend search decoding time is accessing DRAM. 42% of that is spent doing reads and 58% doing writes. The

Table 1: Comparing Software and FPGA-based Hardware

Recognizer Engine	Word Error Rate (%)	Clock (GHz)	Speedup Over Real Time	FOM
Sphinx 3.0	10.88	2.8	3.7	1.32
Hardware: Our recognizer	10.9	0.05	0.5	10

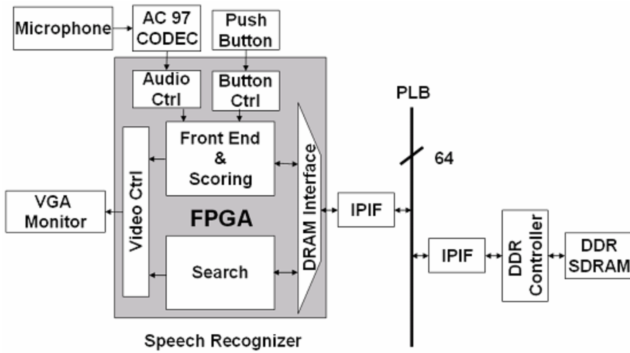

Figure 10. Block diagram of a hardware speech recognition system running on the Xilinx XUP Board

cause for the high percentage of time spent on writes is not due to the quantity of writes, but that only 32-bits are written back at a time, as explained in the previous section.

We evaluate our design by comparing our results with the Sphinx 3.0 software recognizer. Table 1 shows the WER and decoding speed of Sphinx 3.0, running on a 2.8 GHz Xeon processor, and our FPGA-based design running at 50 MHz. While our design does not run faster, one can create a crude *Figure of Merit* (FOM—*larger* is better) for efficiency by dividing the decoder speedup by the clock rate (GHz). Using this metric, our recognizer is 7.6 times *more* efficient than the software based recognizer.

We are also interested in how fast our design could theoretically run without the board's current memory constraints. If we sped up our clock to 200 MHz and directly connect our design to a memory controller with a DDR DRAM with a theoretical bandwidth of 2GB/s, we estimate our design can achieve at least ~3.9 times faster than real-time. If the writeback mechanism is modified to capture 64-bits and burst mode, speedup can increase upwards of 5 times faster than real-time. These results, when compared to Sphinx 3.0's 3.7 times faster than real-time, suggest that if our custom hardware is fitted with dedicated DRAM, it should be able to achieve faster results than a processor running at 15 times its speed.

Figure 11. Fully custom hardware speech recognition system setup

Table 2: Resource utilization breakdown by module

Module	Slices (% of total)	BRAM (% of total)
Acoustic Frontend	3348 (24%)	13 (9%)
GMM Scoring	1004 (7%)	1 (1%)
Backend Search	8802 (67%)	40 (29%)
IO Peripherals	295	8 (6%)

6.2 Final Live-Mode FPGA Speech Recognizer

Once we converged upon the best architecture to implement in hardware, we use synthesizable Verilog to describe the model so it could be mapped on the FPGA of the XUP development board. For the final live-mode hardware speech recognition system, we also wrote peripheral controllers in Verilog for a microphone, VGA monitor, push buttons, and DRAM. At no point do we use the PowerPC cores during decoding. The final block diagram is shown in Figure 10.

Our design is currently completely operational on the Xilinx XUP development board. The final FPGA speech recognition setup can be seen in Figure 11. The core (including peripheral controls) uses 98% of the overall FPGA slices (13449/13696), 45% of the overall 2.44 Mb BRAM (62/136) and ~24Mb of DRAM. The breakdown of each module derived from Xilinx ISE can be seen in Table 2. As stated earlier, we ran the speech recognition core at 50MHz to meet timing requirements while maintaining a reasonable decoding speed. The decoding speed of the FPGA is found to be ~2.3 times slower real-time, which is comparable to the simulator decoding speed. We verify functionality of the FPGA speech recognizer at the bit-level, frame-by-frame, over our entire several-minute data set.

7. CONCLUSION

The Carnegie Mellon *In Silico Vox* project seeks to move best-quality speech recognition technology from its current software-only form into a range of efficient all-hardware implementations. The central thesis is that, like graphics chips, the application is simply too important, too performance hungry, and too power sensitive, to stay as a large software application. To achieve gains in hardware speech recognition, we must first realize the requirements and limitations of a hardware-based recognizer by prototyping the design. We address these issues in this paper and describe in detail the design and implementation of a fully functional speech recognizer on a single Xilinx XUP platform. The design recognizes a 1000 word vocabulary, is speaker-independent, and recognizes continuous (connected) live-mode speech. Our current design runs at 50MHz, decodes at roughly 2.3 times slower real-time, achieves the same accuracy as state-of-the-art software, and is, to the best of our knowledge, the most complex recognizer architecture ever fully committed to a hardware-only form.

Our current work focuses on much larger vocabularies (5000 – 60,000 words), at rates much fast than real-time, leveraging the hardware resources of a more sophisticated FPGA-based platform, the Berkeley BEE2 system [19].

8. ACKNOWLEDGMENTS
This research was supported by the Semiconductor Research Corp., the National Science Foundation, and the MARCO/DARPA Focus Center for Circuit & System Solutions (C2S2). Kai Yu is supported by the NSF Graduate Research Fellowship. The authors would like to thank Richard Stern and Arthur Chan for their valuable suggestions.

9. REFERENCES
[1] Huang, X., Acero, A., and Hon, H., *Spoken Language Processing*: A Guide to Theory, Algorithm and System Development, Prentice Hall PTR, New Jersey, 2001.

[2] "The Talking Cure", The Economist, Mar 12th 2005, p. 11.

[3] Kavaler, R. et al., A Dynamic Time Warp Integrated Circuit for a 1000-Word Recognition System", IEEE Journal of Solid-State Circuits, vol SC-22, NO 1, February 1987, pp 3-14.

[4] Cali, L., Lertora, F., Besana, M., Borgatti, M., "CO-Design Method Enables Speech Recognition SoC," EETimes, Nov. 2001, p. 12.

[5] Mathew, B., Davis, A., and Fang, Z. "A Low-power Accelerator for the SPHINX 3 Speech Recognition System". In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pg 210–219. ACM Press, 2003.

[6] Krishna, R., Mahlke, S., and Austin, T. "Architectural optimizations for low-power, real-time speech recognition". In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 220–231. 2003.

[7] Nedevschi, S., Patra, R., and Brewer, E. "Hardware Speech Recognition on Low-Cost and Low-Power Devices," Proc. Design and Automation Conference, 2005.

[8] Stolzle, A. *et al*. "Integrated Circuits for a Real-Time Large-Vocabulary Continuous Speech Recognition System," *IEEE Journal of Solid-State Circuits*, vol. 26 no. 1, pp 2-11, Jan 1991.

[9] Xilinx Research Labs, *XUP Virtex-II Pro Development System – Hardware Reference Model* Version UG069, 2004.

[10] Pallett, D., "A Look at NIST's Benchmark ASR Tests: Past, Present, and Future", *Proc 2003 IEEE Workshop on Automatic Speech Recognition and Understanding*.

[11] Lin, E., Yu. K., Rutenbar, R., Chen, T. "Moving Speech Recognition from Software to Silicon: the *In Silico Vox* Project" *Proceedings of Interspeech 2006* Sept 2006.

[12] Lin, E., Yu. K., Rutenbar, R., Chen, T. "In Silico Vox: Towards Speech Recognition in Silicon" *HOTCHIPS 18*, August, 2006.

[13] CMU Sphinx Open Source Speech Recognition Engines, http://cmusphinx.sourceforge.net/html/cmusphinx.php.

[14] Huang, X. D., Ariki, Y., and Jack, M. *Hidden Markov Models for Speech Recognition*. Edinburgh University Press, 1990.

[15] Viterbi, A.: "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm." *IEEE Transactions on Information Theory* 13 (1967) 260-269.

[16] Agaram, K., Keckler, S.W., and Burger, D.C. "Characterizing the SPHINX Speech Recognition System," *IBM Austin Center for Advanced Studies Workshop*, January, 2001.

[17] Krishna, R., Austin, T., and Mahlke, S. "Insights into the Memory Demands of Speech Recognition Algorithms," ACM/IEEE 2nd Annual Workshop on Memory Performance Issues, May 2002.

[18] Burger, D. and Austin, T. "The simplescalar tool set version 2.0." Technical Report 1342, Dept of CS, UW, Madison, WI, Jun 1997.

[19] Chang, C., Wawrzynek, J., and Brodersen, R. W. "BEE2: A High-End Reconfigurable Computing System," IEEE Design and Test of Computers, 22(2):114--125, Mar/Apr 2005.