Moving Speech Recognition from Software to Silicon: the *In Silico Vox* Project

Edward C. Lin, Kai Yu, Rob A. Rutenbar, and Tsuhan Chen

Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh PA 15213, USA
{eclin,kaiy,rutenbar,tsuhan}@ece.cmu.edu

Abstract

To achieve much faster decoding, or much lower power consumption, we need to liberate speech recognition from the artificial constraints of its current software-only form, and move the essential computations directly into silicon. There are vast efficiencies waiting to be unlocked in this application – we need the proper architecture to do so. We report results from a first-generation hardware architecture simulated at bit-level, and partial, working FPGA-based prototypes. Simulation results show that rather modest hardware designs, running 10-20X slower than conventional processors, can already decode at 0.6 xRT, running the standard 5K Wall Street Journal benchmark.

1. Introduction

Today's best speech recognizers are all implemented in software, and require a high-end processor and large memory subsystem to achieve real-time decoding. Improving accuracy or decoding speed cannot be achieved without compromising the other. New applications looming on the horizon only exacerbate this problem. For example, applications such as searching large media streams could easily make use of recognizers running 100-1000X faster than realtime. Mobile applications such as cell phones could easily benefit from a reliable voice dictation capability, but have power budgets on the order of 0.1W to accommodate this. A typical general-purpose processor consumes from 10-100W. Although we might hold out hope for better algorithms and tighter implementations, we simply do not see the two to three orders of magnitude improvements in speed, or in energy efficiency, coming from yet more software tweaks.

Thus, we believe it is time to take a serious look at moving speech recognition from software into silicon. We already have one compelling historical precedent: graphics hardware. We simply do not paint pixels in software in any modern computing device: the software solution is too slow, and too power inefficient. Similarly, studies in the DSP community show that custom silicon can be from 100-10,000X more power efficient (computations per unit of power dissipated) than software-programmable solutions [1]. Custom silicon can tailor arithmetic precision to match the demands of the application, deploy as much or as little parallelism as the task warrants, and optimize memory to meet very specific bandwidth needs. A general purpose processor, on the other hand, cannot.

Of course, we are the not the first to consider hardware-based recognizers. There are several earlier efforts [2,3]. However, many were based on algorithms we would today regard as rather primitive. There are also more recent efforts [4,5,6], but these target either low-end applications (e.g., a 30 word vocabulary in [6]), or fail to accelerate other than a few kernels in the overall flow [4], ignoring the complexities inherent in tackling the *entire* problem.

The ***In Silico Vox*** project at Carnegie Mellon is working to design a complete, high-end recognizer in silicon, and understand the difficulties inherent for both very fast, and very low power applications. We employ the CMU Sphinx 3.0 recognizer as our "reference" design. We report our first "hardware-centric" results herein. The paper is organized as follows. Sec 2 offers a new, detailed analysis of the hardware requirements of SPHINX 3.0. Sec 3 summarizes our architecture for fast recognition (we do not consider power issues in this paper). Sec 4 describes two sets of experimental results, from a very detailed cycle-accurate simulation of our design, and from an FPGA-based prototype of portions of our design. Sec 5 offers concluding remarks.

2. Characterizing SPHINX 3.0 for Hardware

Since our goal is to move from software to silicon, we need first to answer two questions: (1) *which* software, and (2) *where* does this software spend its execution time, i.e., what are the challenges? To answer the first question, we choose as a reference model CMU's SPHINX 3.0, a well-known recognizer that uses fully continuous Hidden Markov models (HMM) and flat lexical decoding. Although more recent versions exist, they have sacrificed accuracy for decoding speed, making them unattractive, given our interest in deploying custom hardware to *avoid* performance compromises.

While previous studies [4,7,8] have profiled various speech recognizers, the recognizers used were less sophisticated and/or less accurate. To perform this analysis we first used the Intel VTune performance analyzer [9] to collect performance data while the target program runs on a live processor (an Intel 1.6GHz P4-M, 256 MB RAM). We ran the standard *Broadcast News* task with vocabulary size of 64K words.

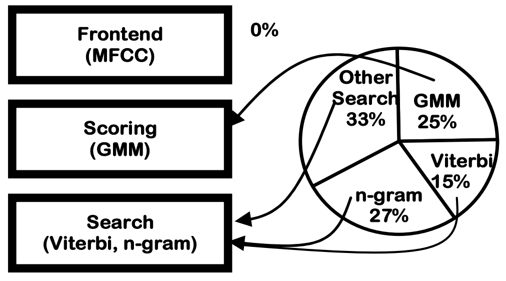

Figure 1 *SPHINX 3.0 execution time breakdown.*

VTune results (see Figure 1) show that roughly 75% of the run-time is spent on search, and 25% on Gaussian mixture model (GMM) scoring. The time spent in the acoustic frontend (from voice to mel-frequency cepstral coefficient (MFCC) feature vectors) is negligible. Thus, in this paper we will primarily focus on GMM computation and search.

A closer look shows memory performance to be the limiting factor, which is consistent with previous studies. SPHINX 3.0 has a large memory footprint and poor cache performance. We demonstrate this by using the SimpleScalar toolkit [10]. We change the size of first-level data cache (DL1) and compare miss rates against the SPEC CPU 2000 benchmarks [11], a set of common software benchmarks.

As shown in Figure 2, with the same cache configuration SPHINX 3.0 has a *significantly* worse DL1 cache miss rate than the average miss rate of the SPEC benchmarks. This can be explained by looking at the speech algorithm. During GMM scoring, constants are continuously streamed in from memory without *any* reuse in a frame. During backend search, data needed for active HMMs (live phone HMMs) comprises the bulk of the memory accesses, and again there is very little reuse within each frame. Given such access patterns, caching the data make little sense unless *all* of the data can be stored. With a memory footprint of beyond 65 MB, this is impossible.

While the majority of the memory accesses have no temporal locality, there are still some data that could benefit from being cached. Figure 2 shows that when the DL1 cache size is increased from 16 KB to 32 KB, the miss rate dramatically falls. We attribute this to caching of tied-state probabilities. Updating active HMM state probabilities requires constantly accessing the tied-state probabilities, so 20 KB (5156 tied-states @ 4 bytes per probability) could be devoted to keeping a frames worth of values readily available. This behavior in DL1 miss rate is present in all the language models we tested.

To determine other important factors related to performance besides memory, we removed memory-related bottlenecks by configuring SimpleScalar to have caches of effectively infinite size, and single cycle latency. Under the new "idealized" conditions, we found that the new performance bottlenecks to be: (1) the number of instructions decoded per cycle and (2) the number of functional units for arithmetic. By doubling the number of instructions fetched and decoded, doubling the number of integer units and quadrupling the number of floating-point units, we decreased the run-time by 30%.

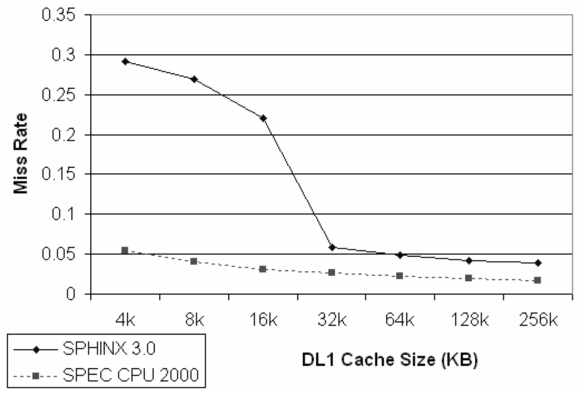

Figure 2: *Miss Rate vs DL1 Cache Size*

These preliminary results show the limitations of using a general processor, and the potential of a custom hardware design. Custom hardware can achieve fine-grained control over memory accesses without the overhead general processors carry. Also, artificial constraints that limit the amount of parallelism exploited, can be eliminated. There seem to be vast efficiencies waiting to be unlocked in this application – we need the right architecture to exploit it.

3. A First-Generation Hardware Architecture

For expediency, we chose a simple set of initial performance goals for our hardware: achieve a decoding rate of roughly 0.5xRT for the 5K word *Wall Street Journal* task, using the *fewest* hardware resources, running at the *slowest* possible clock speed. We use *decoding speed* (xRT) and *word error rate* (WER) as performance metrics. Given limitations of the space, and the fact that Figure 1 clearly shows that the acoustic frontend is of negligible complexity, we focus here on the scoring and backend search tasks.

3.1. Functional Modifications

Functional modifications change the original SPHINX 3.0 algorithm, so they can potentially affect the WER. Only functional modifications that negligibly affected the WER were considered. We briefly survey these in this subsection.

Custom Bit-widths: Custom hardware allows for custom bit-widths, replacing the 32-bit floating-point numbers used throughout the frontend and GMMs with values using fewer bits. Reducing the bit-width reduces the chip area and size of memory required to store the GMM constants. When compared to the original software GMM implementation, this change did not affect the WER, but reduced the average bitwidth by 33%, and decreased memory required to store the GMM constants by 50%.

Log Lookup Table: To compute the log probability of each tied-state in software, the log probability of each GMM mixture is computed, and then summed with the help of a ~100,000 element lookup table. Storing this large lookup table on-chip would be very expensive in hardware, so instead we replace this with a more complex interpolation using four third-order polynomials. At the logic gate, replacing simple lookup and linear interpolation with complex nonlinear interpolation is significantly more area efficient. Using careful nonlinear regression, we also do not affect the WER.

Pruning Threshold: The final major functional change is how the pruning threshold per frame is determined. SPHINX 3.0 first updates the state probabilities of *all* the active HMMs, determines the pruning threshold based on the highest probability present in the frame, and then prunes the active HMMs. This requires passing through the active HMMs *twice*, which practically doubles the memory accesses and hurts performance proportionally. This algorithm also prevents any transition computations from occurring until *after* the second pass occurs, which reduces the effective parallelism. To avoid both bottlenecks, we use the best score from the *previous* frame to determine the pruning threshold. This means an active HMM can immediately be pruned or transitioned after updating all its state probabilities. It also allows for different active HMMs to be doing computation in different stages simultaneously. This method can potentially decrease the number of active HMMs

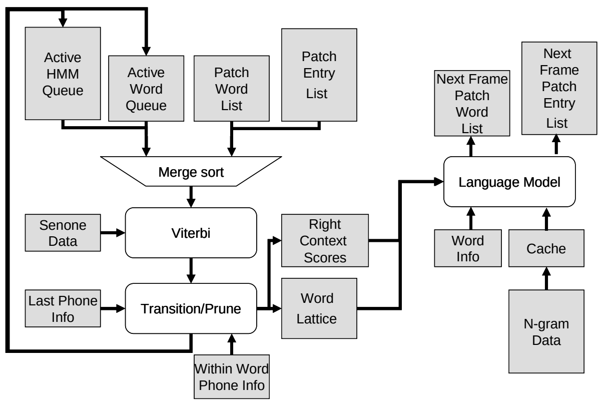

Figure 3: *Search Stage Block Diagram*

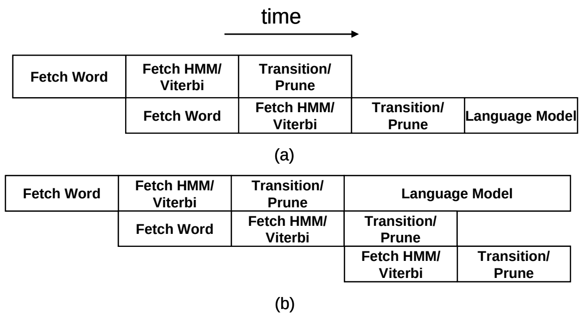

Figure 4: *Pipelines for (a) Fetch Word-Fetch HMM/Viterbi and (b) Language model and Fetch HMM/Viterbi-Transition*

per frame, but through simulation we show there is a negligible increase in WER from 6.707 to 6.725%.

3.2. Structural Modifications

Structural modifications improve hardware performance without affecting the original SPHINX 3.0 algorithm. Most of our present optimization work has been on the backend search stage (e.g., Viterbi search, HMMs, n-gram language model; see Figure 3). In the diagram, square shaded boxes represent memory and all other boxes represent custom logic. We survey the critical structural optimizations here.

Active HMM Storage: In SPHINX 3.0, an HMM requires 40 to 52 bytes of storage, but we compressed this to 28 to 36 bytes without affecting functionality. The active HMMs are also stored consecutively in memory as a queue instead of using linked data structures, which allows for more much regular memory accesses.

Cross-word Transitions: Cross-word transitions normally require fetching the first HMM of the word being transitioned to, and possibly updating the probability of the first state of the first HMM. However, with memory accesses being our major bottleneck, we tried to eliminate this memory access by storing the *all* cross-word probabilities in an on-chip memory called the *Patch List* (see Figure 3). Thus, all cross-word transitions are only compared to the value stored in the on-chip memory, and at the end of every frame the memory stores the most likely cross-word transition per word.

Caching: Our analysis in Section 2 showed that having tied-state probabilities readily available is important for performance, so we have an on-chip memory devoted to it. We also devote an on-chip memory to recently accessed language model probabilities. We found that if an n-gram data structure is accessed once in a frame, then it is likely to be reused, so we created a tiny direct-mapped 64 byte cache to store recent language model probabilities. This simple mechanism gives us a 9% overall speedup.

Pipelining: In the scoring stage, GMM mixture computation is pipelined such that Gaussian probabilities in 6 different dimensions are being computed concurrently. Pipelining not only allows for high data throughput, but also increases the maximum clock frequency at which the design can run. If the data in DRAM can be fetched faster than consumed by a single GMM unit, multiple GMM units can be used to compute separate mixtures in the same frame. Conversely, if the DRAM data cannot be fetched fast enough, multiple GMM units can be used for the same GMM mixture over different frames.

Similarly, in the search phase, we pipeline the decoding process by breaking the stages down according to Figure 4. The Fetch HMM and Viterbi stages are further pipelined. Because each active HMM entry is large, as its data streams in during a fetch, it can be sent to the Viterbi stage for computation. In (a), when an active word with a single active HMM reaches the Fetch HMM/Viterbi stage, the next active word can be fetched. In (b), an active word, active HMM can be performing the cross word transition computation in the n-gram language model, while the next active word is worked on. Active HMMs can continue to decode while the language model is in use, until another active HMM requires the language model at which a stall will occur.

4. Experimental Results

Because designing real silicon is both arduous and expensive, conventional methodology is to build a sequence of detailed simulation and emulation prototypes first. We describe our experience and results with both, in this section.

4.1. Cycle-Accurate Hardware Simulator

To determine both the correctness and effectiveness of a proposed hardware solution, one conventionally implements a high-fidelity model as a software simulator. We need two properties in this simulator: *cycle accurate* and *bit-true*. This means that after every clock tick, all internal state elements of the hardware have exactly the right data values – down to the very last bit – when compared with a “correct” reference model. Our simulator, implemented in C++, achieves this. Although it is common to complain about the CPU demands of tuning/debugging a high-end recognizer, we note that the problem is vastly exacerbated when the recognizer is being *simulated* at bit-level. It was common for ~2 seconds of speech to require 3-4 CPU *days* to simulate.

To estimate the decoding speed of the proposed architecture, we assume a clock frequency of 125 MHz and created a memory model for a commercial synchronous (SDRAM) memory. Both chip and memory clock frequencies are conservative.

Assuming a hardware design with two DRAM ports, one devoted to scoring and the other to search, scoring and search can be performed simultaneously. Thus, we estimate that this hardware design can decode at 0.6xRT. These are untuned results with conservative timing models; we believe we can do much better than this.

Table 1 shows accuracy comparisons between our recognizer – running in simulation – and several software versions of SPHINX running on fast and slow Pentiums. As can be seen, our WER is competitive, our hardware is roughly 3X faster than the fast Pentium, and can run at a clock rate ~8 - 20X slower than the processors. One can create a crude *Figure of Merit* (FOM—*smaller* is better) for efficiency by multiplying clock rate (GHz) by the decoder speedup (xRT). Using this metric, our recognizer is ~10-60 times *more* efficient than software based recognizers.

Table 1: *Comparing Software and (Simulated) Hardware*

Recognizer	WER (%)	Clock (GHz)	Speed (xRT)	FOM
SPHINX 3.3 (fast decoder)	7.32	1.0	1.36	1.36
SPHINX 4 (single CPU)	6.97	1.0	1.22	1.22
SPHINX 4 (dual CPU)	6.97	1.0	0.96	0.96
SPHINX 3.0 (single CPU)	6.707	2.8	1.7	4.76
Hardware Model	**6.725**	**0.125**	**0.60**	**0.075**

4.2. FPGA-based Recognizer Prototype

To further validate our design, we ported it to a small Xilinx field programmable gate array (FPGAs). FPGAs implement gate-level logic in a convenient, rapidly reconfigurable manner. While much less silicon-efficient than a custom chip, they allow for rapid debugging at clock speeds closer to custom silicon. In addition, the FPGA version – being "real hardware" itself – offers yet another concrete datapoint toward the credibility of our overall approach.

We re-implemented our design using the Verilog hardware description language, and again verified we were cycle-accurate and bit-true, frame-by-frame, over our entire several-minute data set. We then synthesize the design to logic level of a Xilinx Virtex-IIPro Xc2VP30 FPGA, using a Xilinx XUP development board. Because this chip is so small – only 200K equivalent logic gates, only 500Mb of on-chip memory -- we used the 1K *Resource Management* task due to resource limitations.

Our design is currently in two separate pieces. We implemented an acoustic frontend module that converts microphone input to MFCC features. The frontend module is a live decoder that updates the cepstral means every 64 frames, exponentially weighting the most recent MFCCs. The acoustic frontend and the full GMM scoring unit are implemented in this piece. The second portion is the full backend search, the Viterbi/HMM/n-gram computations. Word hypotheses display on a VGA monitor.

As of this writing, the two individual pieces are working successfully, with the DRAM interface running at 100MHz, the frontend/scoring unit running at 50MHz, and the backend search running at 100MHz. At these relatively modest speeds, we should decode roughly 1.2xRT. Our current challenge is the very small size of this FPGA, on which both pieces of the design cannot currently fit. Thus, we are now in the process of moving to a much larger FPGA-based emulation platform, the Berkeley BEE engine [12], which consists of 5 large FPGAs, roughly 5M equivalent gates, and 16GB of available memory. We believe this will let us move up to larger (50K+ word) faster recognizers.

5. Conclusions

We have argued that the time is ripe to migrate speech recognition from software to hardware, to liberate it from the artificial constraints on speed / power made by general purpose CPUs. From detailed performance studies of the CMU SPHINX 3.0 system, we proposed a first-generation hardware architecture, developed fully detailed hardware models, and successfully prototyped individual pieces of this design in FPGA form. We are a long way from where we believe a silicon-centric approach can take us, but preliminary results are very promising.

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

6. References

[1] Brodersen, R.W. "Low-Voltage Design for Portable Systems," Panel Session Presentation, International Symposium on Solid State Circuits (ISSCC), Feb. 2002.

[2] 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.

[3] Hon, H. W, A Survery of Hardware Architectures Designed for Speech Recognition. Technical Report CMU-CS-91-169, 1991.

[4] 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.

[5] 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. ACM Press, 2003.

[6] Nedevschi, S. Patra, R. and Brewer, E. "Hardware Speech Recognition on Low-Cost and Low-Power Devices". To Appear in Design and Automation Conference, June 2005

[7] 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.

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

[9] Intel Performance Analyzers Homepage. http://developer.intel.com/software/products/vtune/index.htm.

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

[11] Cantin, J. and Hill, M. "Cache Performance for Selected SPEC CPU2000 Benchmarks", Computer Architecture News (CAN), September 2001.

[12] 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.