A LOW-POWER, FIXED-POINT, FRONT-END FEATURE EXTRACTION FOR A DISTRIBUTED SPEECH RECOGNITION SYSTEM

Brian Delaney[*], *Nikil Jayant*

School of Electrical and Computer Engineering
Multimedia Communications Lab
Atlanta, GA 30332

Mat Hans, Tajana Simunic, Andrea Acquaviva

Hewlett-Packard Laboratories
Client and Media Systems Lab
1501 Page Mill Road
Palo Alto, CA 94304

ABSTRACT

This work describes the optimization of a signal processing front-end for a distributed speech recognition system with the goal of reducing power consumption. Two categories of source code optimizations were used, architectural and algorithmic. Architectural optimizations reduce the power consumption for a particular system, in this case, the HP Labs Smartbadge IV prototype portable system. Algorithmic optimizations are more general and involve changes in the algorithmic implementation of the source code to run faster and consume less power. A cycle accurate energy simulation shows a reduction in power usage by 83.5% with these optimizations. The optimized source code runs 34 times faster than the original code, therefore it can run at lower processor clock speeds and voltages for further reductions in power consumption. This technique, known as dynamic voltage scaling, was implemented on the Smartbadge IV hardware for an overall reduction in power usage of 89.2%.

1. INTRODUCTION

This work describes the optimization of a signal processing front-end feature extraction for a distributed speech recognition system. The baseline system used in the experiments is version 0.3 of the open-source Sphinx II speech recognizer from Carnegie Mellon University [1]. The optimization methods used for the algorithm substantially decrease the power usage while increasing speed (measured in processor cycle counts). Estimates of total power usage are performed using a cycle-accurate energy consumption simulator [2]. The architecture of the embedded system simulated in the experiments mimics that of the Smartbadge IV system developed at the Appliance Platform department of HP Labs [3]. In addition to performing energy consumption simulations to evaluate the quality of source code optimizations, we also implemented and ran the optimized version of the front-end on Smartbadge IV hardware. We found that real-time signal processing of speech is possible at eleven discrete CPU frequency and voltage settings, thus enabling further power savings.

A block diagram of a speech recognition system is shown in Figure 1. It can easily be divided into two parts, a front-end and a back-end. The front-end produces a set of acoustic observations which are useful in recognizing speech. The back-end is where most of the computation and memory usage takes place. The back-end can easily use hundreds of Mbytes of memory and hundreds of MIPS of computation.

[*]Brian Delaney performed the work while at HP Labs.

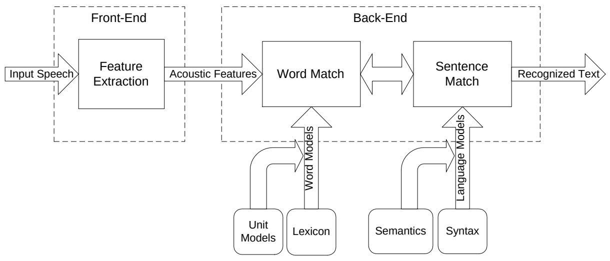


Fig. 1. Block diagram of a typical speech recognizer [4].

Since the front-end feature extraction step is relatively low in complexity, it is desirable to perform this step on the embedded device and to send compressed features across the network. It has been shown that these features can be compressed with little effect on the error rate of the speech recognizer [5]. The ETSI standard for distributed speech recognition describes algorithms to compute, compress, and transmit these speech features [6]. We are considering only the computation of these features and not the compression and transmission.

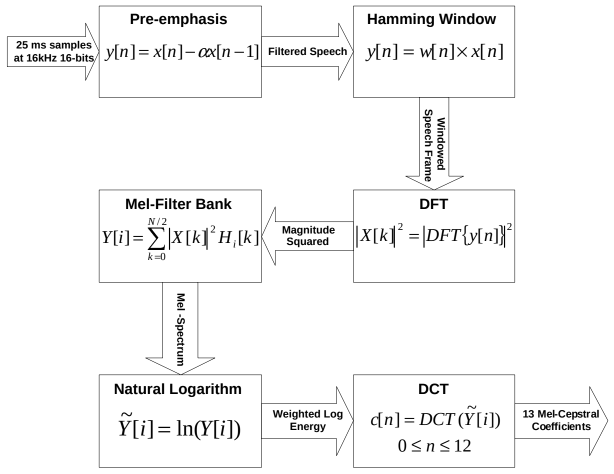
2. THE SIGNAL PROCESSING FRONT-END

The acoustic observations generated by the signal processing front-end or "feature extraction" step are mel-frequency cepstral coefficients. Mel-frequency cepstral coefficients are calculated using the real cepstrum, defined as the inverse Fourier transform of the log spectrum:

$$c_s(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \log|S(\omega)|\, e^{j\omega n} d\omega \tag{1}$$

where $S(\omega)$ is the spectrum of the speech signal. The features used in the Sphinx II speech recognizer consist of 13 mel-frequency cepstral coefficients computed every 10ms. Secondary features, consisting of first and second time derivatives of the cepstrum, are also used, but they can be calculated easily at the back-end. A more in depth discussion of the theory and properties of the cepstrum can be found in [7].

In practice, the mel-frequency cepstral coefficients can be computed using the algorithm in Figure 2. We assume the digitized speech is 16-bit linear sampled at 16kHz. A pre-emphasis filter whitens the speech signal and overlapping frames of 25ms are



25 ms samples at 16kHz 16-bits
Pre-emphasis
$y[n] = x[n] - \alpha x[n-1]$
Filtered Speech
Hamming Window
$y[n] = w[n] \times x[n]$
Windowed Speech Frame
DFT
$|X[k]|^2 = |DFT\{y[n]\}|^2$
Magnitude Squared
Mel-Filter Bank
$Y[i] = \sum_{k=0}^{N/2} |X[k]|^2 H_i[k]$
Mel-Spectrum
Natural Logarithm
$\tilde{Y}[i] = \ln(Y[i])$
Weighted Log Energy
DCT
$c[n] = DCT(\tilde{Y}[i])$
$0 \le n \le 12$
13 Mel-Cepstral Coefficients

Fig. 2. The algorithm used to compute the mel-cepstrum.

multiplied by a Hamming window. Next the magnitude squared of the discrete Fourier transform (DFT) is computed. The magnitude squared is processed by a set of mel-filter banks to produce an estimate of the mel-spectrum. The mel-filter banks are implemented as a series of overlapping triangle filters, $H_i[k]$, that are centered on equally spaced frequencies in the mel-scale. The result is an estimate of the total energy in the ith critical band:

$$Y[i] = \sum_{k=0}^{N/2} |X[k]|^2 H_i[k] \qquad (2)$$

where $X[k]$ is the DFT of the windowed speech signal and $H_i[k]$ contains the filter-bank coefficients. Finally, the logarithm of the mel-spectrum is taken to produce a weighted log energy, $\tilde{Y}[i]$. The weighted log energy is real and even, so the inverse Fourier transform can be implemented as a discrete cosine transform (DCT) with equivalent results.

3. LOW-POWER OPTIMIZATION

Implementing the front end feature extraction for a distributed speech recognition system on an embedded platform requires not only speed, but also power optimization, since the battery lifetime in such devices is very limited. This work discusses both the source-code and the run-time optimizations. A good overview of the previous work done in code and run-time optimization for low power is presented in [13].

The source code optimizations can be grouped into two categories. The first category, architectural optimizations, aims to reduce power consumption while increasing speed by using optimization methods targeted to a particular processor or platform (e.g. an embedded system with no floating-point hardware). Ideally, many of these optimizations should be done by a compiler. However, currently available compilers for most embedded systems do not have these optimizations built-in. In addition, measurements presented in [2] show that the improvements that can be gained using standard compiler optimizations are marginal compared to writing energy efficient source code. The second category of source code optimizations is more general and involves changes in the algorithmic implementation of the source code with the goal of faster performance with less power consumption.

The final optimization presented in this work, dynamic voltage scaling (DVS), is the most general since it can be applied at run-time without any changes to the source code. Dynamic voltage scaling algorithms reduce energy consumption by changing processor speed and voltage at run-time depending on the needs of the applications running. The maximum power savings obtained with DVS are proportional to the savings in frequency and to the square of voltage.

3.1. Architectural Optimization

Signal processing algorithms such as the one in Figure 2 are generally mathematically intensive, therefore a significant amount of effort was spent in optimizing the arithmetic. In addition, simple C code optimizations were employed to help the compiler generate more efficient code [9].

Profiling of the source code on a StrongARM simulator revealed that over 90% of the time was spent in floating-point emulation. The StrongARM has no on-chip floating-point processor, so all floating-point operations must be emulated in software. Simply changing from double to single precision floats improved the performance considerably. However, profiling showed that 80% of the time was still being spent in floating point emulation. Any further gains require fixed-point arithmetic.

Fixed-point arithmetic uses scaled integers to perform basic math functions using the existing integer hardware. The scaling factor (or location of the decimal point) is fixed at design time and is designated by Qn, where n is the number of bits to the right of the decimal. The basic rules of arithmetic still hold; adding two numbers requires that the decimal points must line up. Multiplying two numbers in Qn format yields a number in $Q2n$ format.

Implementing a pre-emphasis filter and Hamming window using fixed-point arithmetic is straight-forward. Fixed-point FFTs are well studied and have often been implemented on digital signal processor chips.

After passing the input frame through the FFT, the mel filter bank must be applied. Recall that the filter bank amplitudes in (2) are calculated using the squared magnitude. This presents some challenges since this squared number multiplied by the filter coefficients, $H_i[k]$, can easily overflow the 32-bit registers. A 64-bit result can be obtained from the StrongARM multiplier using assembly language, but overflow can be avoided simply by rewriting the filter bank equation (2) to use just the magnitude:

$$Y[i] = \sum_{k=0}^{N/2} \left(|X[k]| \sqrt{H_i[k]} \right)^2 \qquad (3)$$

This avoids overflow since $H_i[k] \ll 1$, therefore the result of each multiplication is small. The coefficients, $\sqrt{H_i[k]}$, are stored in a lookup table.

The one drawback to this method is that computing the magnitude requires a square root operation. Fast integer square root algorithms exist, but they must be used on each output from the FFT, which is costly. Fortunately, the magnitude can be estimated as a linear combination of the real and imaginary parts using the following equation [11]:

$$|x| \approx \alpha \max(|\Re\{x\}|, |\Im\{x\}|) + \beta \min(|\Re\{x\}|, \Im\{x\}|) \qquad (4)$$

where α and β are chosen to minimize a particular kind of error, and $\Re\{x\}$ and $\Im\{x\}$ represent the real and imaginary parts of the complex number x. This formula rotates a complex phasor to between 0 and $\pi/4$ radians and then takes a linear combination of the real and imaginary parts. The values of α and β are chosen to have an easy fixed-point representation that minimizes the mean error.

Computing the first 13 coefficients of the DCT is relatively easy to do in fixed-point arithmetic, but taking the natural logarithm is a more difficult task. However, there is an interesting algorithm to estimate $\log_2(x)$ using simple bit manipulation, which is faster than other methods of calculating the logarithm. This algorithm, described in [12, 13], is very low in complexity and gives an approximate fixed-point result. The $\ln(x)$ can be determined by multiplying by a constant as follows:

$$\ln(x) = \log_2(x)\ln(2) \tag{5}$$

One final adjustment must be made when x is itself a fixed-point number in Qn format, which is just a scaled integer:

$$\ln\left(\frac{x}{2^n}\right) = [\log_2(x) - \log_2(2^n)]\ln(2) \tag{6}$$

$$\ln\left(\frac{x}{2^n}\right) = [\log_2(x) - n]\ln(2) \tag{7}$$

Equation (7) is the expression used to calculate the natural log of a fixed-point number. Using precision of Q3, this estimate of the logarithm has a maximum error of around 0.152 and an average error of around 0.0866.

3.2. Algorithmic Optimization

Profiling of the original source code under a StrongARM simulator revealed that most of the execution time was spent in the computation of the DFT (which is implemented as an FFT). Since speech is a real-valued signal, an N-point complex FFT can be reduced to an $N/2$-point real FFT [10]. Some further processing of the output is required to get the desired result, but this overhead is minimal compared to the reduction in computation. Additional savings can be obtained when the trigonometric functions used in the computation of the FFT are pre-computed and stored in a lookup table, thus eliminating multiple function calls in the FFT loop.

3.3. Dynamic Voltage Scaling

Once the code is optimized for both power consumption and speed, further savings are possible by changing the processing frequency and voltage at run-time. In this work, we investigate the savings possible with DVS for the front-end of a speech recognizer running on Smartbadge IV hardware. The StrongARM processor on Smartbadge IV can be configured at run-time by a simple write to a hardware register to execute at one of eleven different frequencies. Note that the number of frequencies is predefined by the design of the StrongARM processor. We measured the transition time between two different frequency settings at 150 microseconds. Since typical processing time for the front-end is much longer than the transition time, it is possible to change the CPU frequency without perceivable overhead. For each frequency, there is a minimum voltage the SA-1110 needs in order to run correctly, but with lower energy consumption. The easiest way to determine the lowest possible frequency and voltage for such stand alone application is to run it at all possible frequency settings, with voltage set to minimum allowed, and observe if the code still runs in real time. In our case, we obtained real time performance at all possible frequency and voltage settings.

4. RESULTS

Three main criteria are considered in order to evaluate the effectiveness of a particular optimization: performance (in terms of processor cycle count), energy consumption, and accuracy or word error rate (WER). Simulation results for processing one frame (25ms) of speech on the Smartbadge IV architecture running at 202.4 MHz are shown in Figure 3. The x-axis shows the source code in various stages of optimization. The “baseline” source code contains no software optimizations. The “optimized float” code contains the set of optimizations described in section 3.2 as well as some of the C source optimizations described in [9]. Double precision floating-point numbers were changed to single precision 32-bit floats in the “32-bit float” version of the code. Finally, the “fixed-point” implementation contains all of the source code optimizations described in this paper. For each version of the code, we report the performance (in CPU cycles) and the total battery energy consumed (in μWhrs). The simulation results are computed by the cycle-accurate energy simulator, and include processor core and level 1 cache energy, interconnect and pin energy, energy used by the memory, losses from the DC/DC converter, and battery inefficiency [2]. The reduction in energy consumption is not as dramatic as the performance improvement for the fixed-point version due to an increase in memory references per unit time. In fixed-point code, basic math operations are reduced to a few cycles as opposed to long iterations of floating-point emulation which do not require as many memory references. However, we have still achieved a reduction in the total battery energy required to process one frame of speech data by 83.5%.

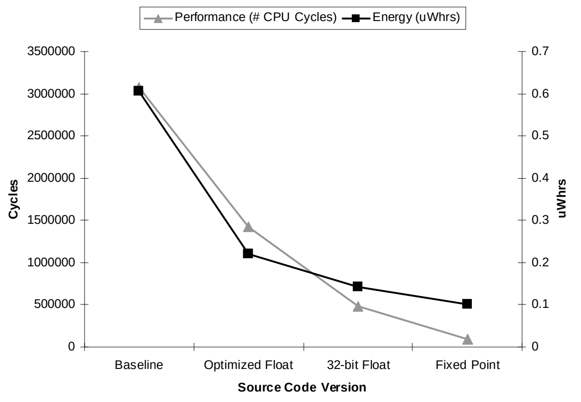


Fig. 3. Performance and energy consumption per frame of speech.

The front-end was tested using the TIDIGITS speech database, and the results are shown in Table 1. A continuous digit speech recognizer was trained using the TIDIGITS database of 8,623 utterances from both male and female speakers. The original floating point front-end was used to generate mel-frequency cepstral coefficients for the training set. No secondary features (first and second time derivatives of the mel-frequency cepstrum) were used in the training or test phases. The trained speech models were then used to recognize speech from the TIDIGITS test set of 8,700 utterances. The WER was calculated using the various front-end implementations and is shown in Table 1. There is no loss in accuracy among the three floating-point implementations, but the fixed-point implementation uses some approximate algorithms which can create a slight mismatch between the training and test data. We were able to eliminate the slight 0.1% increase in WER by using the fixed-point front-end during the training phase. In addition, Table 1 shows a minimal increase in lookup table size and code size, so the memory requirements for the fixed-point optimized code are about the same. Another performance metric reported in Table 1 is how long it took for each code implementation to process 1 second of speech at the processor clock speed of 202.4 MHz (Time column). The fixed-point version runs 34 times faster than the baseline system.

Table 1. TIDIGITS test set results.

	Code size (Bytes)	Lookup table (Bytes)	Time (sec)	WER %
Baseline	29704	N/A	1.510	4.2%
Optimized Float	31960	88120	0.699	4.2%
32-bit Float	31272	88120	0.235	4.2%
Fixed-Point	33124	88136	0.043	4.3%

Because the fixed-point code runs much faster than real-time at 202.4MHz, it is possible to get further reductions in power usage by using DVS as discussed in section 3.3. The results from this experiment are shown in Table 2. These power measurements are performed on the Smartbadge IV system running the eCos embedded operating system and using the WaveLAN card to transmit the uncompressed cepstral parameters. The P_{sys} measurement is taken from the main power supply output. At 59 MHz the algorithm still runs in real-time, and the system uses 34.7% less power than at 206 MHz. Combining the DVS results with the source code optimizations, we calculate the overall reduction in power consumption to be 89.2%.

Table 2. Measured Power Consumption with DVS.

Frequency (MHz)	Voltage (V)	P_{sys} (mW)
59	0.78	1721
74	0.94	1807
89	1.09	1901
103	1.21	2029
118	1.33	2114
132	1.42	2234
147	1.51	2320
162	1.57	2432
176	1.63	2508
191	1.67	2568
206	1.69	2636

5. CONCLUSION

In this paper, we have outlined some optimization techniques to reduce the energy consumption of a particular signal processing algorithm. On embedded systems with no floating-point hardware, fixed-point arithmetic is an important step in lowering the power consumption of a program. However, careful attention must be paid to basic math functions (i.e. cosine, log, etc.) and overflow/underflow issues. Approximate algorithms perform well for certain applications and can result in huge savings in both time and power usage. By using software optimizations, we were able to achieve a reduction in energy usage by 83.5% compared to the unoptimized source code. Finally, we show that additional power savings are possible by scaling processor frequency and voltage at run time, while still meeting the performance requirements. At the lowest frequency/voltage setting, we calculate an overall reduction in power consumption by 89.2%.

6. ACKNOWLEDGMENTS

The authors would like to thank Fred Kitson and Mark Smith of HP Labs for their help. This work was supported by the Hewlett-Packard Laboratory.

7. REFERENCES

[1] Carnegie Mellon University, “Sphinx-II ASR system (open-source version 0.3),” http://www.speech.cs.cmu.edu/speech/.

[2] T. Simunic, L. Benini, and G. De Micheli, “Energy-efficient design of battery-powered embedded systems,” *Special Issue of IEEE TVLSI*, pp. 18–28, May 2001.

[3] G. Q. Maguire, M. Smith, and H. W. Peter Beadle, “Smart-badges: A wearable computer and communication system,” 6th International Workshop on Hardware/Software Codesign, 1998, Invited Talk.

[4] Lawrence R. Rabiner, “Applications of speech recognition to the area of telecommunications,” in *1997 IEEE Workshop on Automatic Speech Recognition and Understanding Proceedings*, 1997, pp. 501–510.

[5] Qifeng Zhu and Abeer Alwan, “An efficient and scalable 2d dct-based feature coding scheme for remote speech recognition,” in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2001.

[6] “Speech processing, transmission and quality aspects (stq); distributed speech recognition; front-end feature extraction algorithm; compression algorithms,” ETSI Standard: ETSI ES 201 108 v1.1.2, 2000, http://www.etsi.org/stq.

[7] Deller, Proakis, and Hansen, *Discrete–Time Processing of Speech Signals*, Prentice Hall, Upper Saddle River, NJ, 1987.

[8] Y. Li and J. Henkel, “A framework for estimateing and minimizing energy dissipation of embedded hw/sw systems,” in *Processings of DAC 1998*, 1998, pp. 188–193.

[9] “Writing Efficient C for ARM,” Application Note 34, Jan. 1998, ARM Inc.

[10] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*, Cambridge University Press, Cambridge England, 1992.

[11] Marvin E. Frerking, *Digital Signal Processing in Communications Systems*, Van Nostrand Reinhold, 1994.

[12] Jack W. Crenshaw, *Math Toolkit for Real-Time Programming*, CMP Books, Lawrence, Kansas, 2000.

[13] B. Delaney, M. Hans, T. Simunic, A. Aquaviva, “A Low-Power, Fixed-Point Front-End Feature Extraction for a Distributed Speech Recognition System,” *HP Technical Report*, HPL-2001-252, 2001.