A High-speed, Low-Resource ASR Back-end Based on Custom Arithmetic

Xiao Li, Jonathan Malkin, Jeff Bilmes

Abstract— **With the skyrocketing popularity of mobile devices, new processing methods tailored to a specific application have become necessary for low-resource systems. This work presents a high-speed, low-resource speech recognition system using *custom arithmetic* units, where all system variables are represented by integer indices and all arithmetic operations are replaced by hardware-based table lookups. To this end, several reordering and rescaling techniques, including two accumulation structures for Gaussian evaluation and a novel method for the normalization of Viterbi search scores, are proposed to ensure low entropy for all variables. Furthermore, a discriminatively inspired distortion measure is investigated for scalar quantization of forward probabilities to maximize the recognition rate. Finally, heuristic algorithms are explored to optimize system-wide resource allocation. Our best bit-width allocation scheme only requires 59kB of ROMs to hold the lookup tables, and its recognition performance with various vocabulary sizes in both clean and noisy conditions is nearly as good as that of a system using a 32-bit floating-point unit. Simulations on various architectures show that on most modern processor designs, we can expect a cycle-count speedup of at least 3 times over systems with floating-point units. Additionally, the memory bandwidth is reduced by over 70% and the offline storage for model parameters is reduced by 80%.**

Index Terms— **Speech recognition, custom arithmetic, high speed, low resource, quantization, normalization, bit-width allocation, discriminative distortion measure, forward probability normalization and scaling, alpha recursion**

I. Introduction

THE burgeoning development of mobile devices has brought about a great need for a more friendly and convenient user interface. Automatic speech recognition (ASR) has unquestionable utility when used in environments without a keyboard, or where hands are unavailable. To take full advantage of the envisioned "smart home" of the future, in which most appliances are online and connected, we will want a fully-functioning ASR system on a highly portable device, such as a watch, necklace, or pendant. However, unlike desktop applications with ample memory and a perpetual power supply, portable devices suffer from limited computational and memory resources and strict power consumption constraints. Most state-of-the-art ASR systems running on desktops use continuous-density HMMs (CHMM with floating-point arithmetic. These systems are computationally expensive, posing potential problems for real-time processing and battery life. The development of a high-speed, low-resource ASR system, therefore, becomes crucial to the prevalence of speech technologies on mobile devices.

This work was funded by National Science Foundation under Grant Award-0086032. The authors are with the Department of Electrical Engineering, University of Washington. Emails: {lixiao, jsm, bilmes}@ee.washington.edu

In the literature, there are many techniques to speed up computation at the algorithmic level, among which quantization with table lookups has been extensively used. First, observation vectors or sub-vectors can be quantized and their state or Gaussian mixture component likelihoods can be obtained efficiently via pre-computed tables. A discrete-density HMM, for example, applies vector quantization (VQ) to the observations and approximates the state likelihood computation by lookup operations. As a further improvement, a discrete mixture HMM assumes discrete distributions at the scalar or sub-vector level of a mixture model, and applies scalar quantization or sub-vector quantization to the observations [1], [2]. Even in a CHMM, the computational load can be greatly reduced by restricting the precise likelihood computation to the most relevant Gaussians using VQ [3], [4]. Second, quantization techniques also contribute to a compact representation of model parameters, which not only saves memory but also reduces computational cost [5]–[9].

The problem can also be approached from the hardware side. A floating-point unit is power-hungry, and requires a rather large chip area when implemented. Software implementation of floating-point arithmetic takes less power and chip area, but has significantly higher latencies [10]. Additionally, speech recognizers usually do not use the precision of a floating-point representation efficiently. Fixed-point arithmetic offers only a partial solution. Operations can be much faster using a fixed-point implementation [11]–[13]. But this method often cuts the available dynamic range without having its representation precision fully utilized. Additionally, some operations can still take numerous processor cycles to complete. Fixed-point ALUs, therefore, are often combined with rescaling and table lookup techniques for better performance.

With 32-bit computing having reached the embedded market and after years of finding ways to make general purpose chips more powerful, the use of custom logic might seem a rather curious choice [14]. Many signal processing applications produce system variables (system inputs, outputs and all intermediate values) with very low entropy. It would be beneficial to "record" these computation results so that they may be reused many times in the future, thereby amortizing the cost of computation. [15] uses cache-like structures they call memo-tables to store the outputs of particular instruction types. It performs a table lookup in parallel with conventional computation which is halted if the lookup succeeds. The paper argues that the cycle time of a memo-table lookup is comparable to that of a cache lookup.

Generally speaking, quantization on a lower-entropy variable, using a fixed number of bits, has a smaller expected quantization distortion. Motivated by empirically observed low

entropy of the variables within several speech recognition systems, we present a novel ***custom arithmetic*** architecture based on high-speed lookup tables (LUTs). Therein, each system variable is quantized to low precision and each arithmetic operation is pre-computed for each of its input and output codewords vectors. The goal is appealing, considering the high speed and low power consumption of a hardware-based lookup compared to that of a complicated arithmetic function. On the other hand, the objective also looks daunting, since an ASR system might have dozens of variables and operations, leading to a prohibitive amount of storage for tables. Therefore, to implement an ASR system using custom arithmetic units, the first and foremost assumption is that the value distributions of such a system have entropy low enough for low-precision quantization. Second, the quantization of a specific variable should ideally be consistent with minimizing the degradation in recognition performance. Finally, a bit-width allocation algorithm must be provided to optimize the resource performance. While in [16] and [17] we proposed a general design methodology for custom arithmetic and reported preliminary results for system development, this paper approaches the problem systematically, discusses the solutions in great detail and presents new evaluation results with different vocabulary sizes and in different noisy conditions.

We choose to apply custom arithmetic to the back-end but not to the front-end for several reasons. The back-end accounts for most of the computational load of an ASR system, but it has fewer variables than the front-end since many of its operations are repetitive. By contrast, the front-end has relatively low computational cost but a large variety of variables and operations which would quickly complicate the lookup table design. In addition, the fixed-point arithmetic for feature extraction has been well studied and can be implemented by DSPs very efficiently [18]. Therefore, we envision a chip using a combination of both standard fixed-point arithmetic for the front-end, and custom arithmetic for the back-end.

The rest of the paper is organized as follows. Section II discusses the general mechanism of custom arithmetic. Section III and Section IV present our computation reordering technique for Gaussian evaluation and our normalization method for Viterbi search respectively. Section V formulates a discriminatively inspired distortion measure for quantizing forward probabilities. Section VI investigates several heuristics for bit-width allocation. Section VII describes our system organization, and Section VIII reports our word error rate (WER) and cycle time experiments and results, followed by concluding remarks.

II. General Design Methodology

In this section we present an ASR system driven by custom arithmetic, where all floating-point calculations are precomputed. In addition, we address several potential issues associated with custom arithmetic design for an ASR system.

A. General Arithmetic Mechanism

A high-level programming language allows complex expressions involving multiple operands. We split all such complex expressions into sequences of two-operand operations by introducing intermediate variables. We can then express any operation on *scalar* variables V_i and V_j, with the result saved as V_h, by a function, $V_h = F_k(V_i, V_j)$, where $i, j, h \in \{1..L\}$. The function $F_k(\cdot)$, $k \in \{1..K\}$, can be an arbitrary arithmetic operation or sequence of operations, e.g. $V_h = (V_i - V_j)^2$.

The first step of custom arithmetic design is to create a codebook for each scalar variable. The codewords are the quantized values of that variable, and the indices of those codewords are consecutive integers. For a variable with value $V_l = x$, the closest codeword in the codebook is denoted as $Q_{V_l}(x)$, and its associated index is denoted as $I_{V_l}(x)$.

Second, a table T_{F_k} is created to store all allowable values for the function $F_k(\cdot)$, as defined by the input and output codebooks. Each address in the table is determined by the indices of the input operand codewords and the output is the index of the result's codeword. Equationally, for $z = F_k(x, y)$, we have $I_{V_h}(z) = T_{F_k}(I_{V_i}(x), I_{V_j}(y))$. If the output and the two inputs have bit-widths of n_0, n_1, and n_2, respectively, then the table requires a total storage of $n_0 \cdot 2^{n_1+n_2}$ bits.

The final step in designing this custom arithmetic system is to replace all floating-point values with the corresponding integer indices and approximate all two-operand arithmetic operations with table lookups. Note that the output index is used as the input of the next table lookup, so that all data flow and storage are represented in integer form, and all complex operations become a series of simple table accesses.

The physical device realization of the LUTs is beyond the scope of this work. The implementation of custom arithmetic units can be simplified using reconfigurable logic [19], [20], although at the expense of increased power consumption.

B. Design Issues for ASR

In spite of the attractiveness of custom arithmetic, such a system becomes unrealistic if the table size gets too large. This poses several challenges to the custom arithmetic design for an ASR system:

1) How to modify the decoding algorithm to ensure low entropy for all system variables.
2) What are quantization methods consistent with recognition rate maximization.
3) How to allocate bit-widths among system variables to optimize resource performance.

As stated in the introduction, the foremost assumption of a custom arithmetic based system is the low entropy of all system variables. Most variables in a state-of-the-art ASR back-end, as will be seen in Section VIII, can be quantized to low precision without loss of recognition accuracy. However there do exist several variables with high entropy, which must be tackled by algorithmic level modification. In the Mahalanobis distance calculation of Gaussian evaluation, for example, the distance is accumulated along the dimension of the features, resulting in a relatively spread-out distribution covering all partial accumulations. Additionally, the forward probability α in decoding possesses a potentially more fatal problem — the forward pass computes over an arbitrarily long utterance in real applications, making α's value distribution unknown to the quantizer at the codebook design stage. Consequently,

the designed codebook may not cover all values that may occur at decode time, leading to poor recognition performance. Although certain normalization techniques have been proposed in the literature, they can not essentially solve the problem for custom arithmetic design. Potential solutions to this first issue will be discussed in Section III and Section IV.

The second issue is in fact a quantization problem. Since the bit-width of each variable directly influences the table size, we want each variable to be scalar quantized to as low precision as possible without degradation in recognition rate. The distortion measure should ideally be consistent with minimizing recognition degradation. In this work, we are particularly interested in further compressing the forward probability α, which has the highest entropy even after rescaling, as will be shown in Section VIII. Section V presents a discriminatively inspired distortion measure to quantize this variable.

Finally, the last issue is a search problem for optimally allocating memory resources among tables. Since the search space could be quite large for an ASR system, we investigate several heuristics in Section VI to find the best search scheme within the existing computational capacity.

III. Computation Reordering for Gaussian Likelihood Evaluation

Gaussian evaluation can dominate the operational load by taking up to 96% of the total computation for a typical small vocabulary application [3], and 30% to 70% of the total computation for LVCSR tasks [4]. However, the nature of Gaussian evaluation makes this task particularly suited to our custom arithmetic, as will be seen in this section. As a side note, the required memory footprint for these calculations can also be reduced through the use of lookup tables, providing an added benefit.

A. Problem Formulation

Log arithmetic is widely used in practical ASR systems to achieve numerical values with a very wide dynamic range. In this paper, a variable with a bar denotes its log value. For example, $\bar{x} = \log x$. Also, $\oplus$ denotes log addition where $\bar{x} \oplus \bar{y} = \log(e^{\bar{x}} + e^{\bar{y}})$[1]. To this end, the log state likelihood $\bar{b}_j(t)$ of the t^{th} observation vector $(x_1(t), x_2(t), ..., x_D(t))$, given a certain state $q_t = j$, can be expressed as

$$\bar{b}_j(t) = \bigoplus_{i \in M_j} (\bar{w}_i + c_i - \frac{1}{2} \sum_{k=1}^{D} \frac{(x_k(t) - \mu_{i,k})^2}{\sigma_{i,k}^2}), \quad (1)$$

where M_j is the subset of diagonal Gaussians belonging to state j; the variables $\mu_{i,k}$ and $\sigma_{i,k}$ are the mean and variance scalars of a Gaussian respectively; $\bar{w}_i$ is the log value of the i^{th} component responsibility (or mixture weight) and c_i is a constant, both of which can be computed offline.

As mentioned earlier, many operations in Gaussian evaluation are repetitive: to evaluate the observation probabilities for an utterance with T frames in a system with M different Gaussian components, the operation of the form $(x_k(t) - \mu_{i,k})^2/\sigma_{i,k}^2$ will be performed $T \times M \times D$ times, which easily could indicate millions of floating-point multiplications. Similarly, the more expensive log addition may be performed thousands of times. Substituting in simple LUTs can provide substantial benefit.

A crucial problem inherent to Gaussian likelihood evaluation is that there are two iterative operations suggested by (1). One is $e_i(t) \triangleq \sum_{k=1}^{D} d_{i,k}(t)$, where $d_{i,k}(t) \triangleq (x_k(t) - \mu_{i,k})^2/\sigma_{i,k}^2$, and the other is $\bar{b}_j(t)$ associated with the log addition. The accumulations associated with these operations may produce high-entropy variables, making codebook design difficult. For example, in the above calculation, consider the distribution of the partial accumulation at different points in the summation calculation. Using a temporary value to store the result of the partial accumulation, we initially have $tmp = d_{i,1}(t) + d_{i,2}(t)$. If we assume that the $d_{i,k}(t)$ values are identically distributed, then tmp at this point will have a dynamic range of twice that of $d_{i,k}(t)$. At the next step, $tmp = tmp + d_{i,3}(t)$, tmp will have a dynamic range of three times that of $d_{i,k}(t)$, and so forth.

Before moving on, there is a subtle but important distinction to make between the entropy and the dynamic range of a random variable. These two concepts are related in accumulation. If all input values to be summed are identically distributed, an accumulation will see many partial accumulative values at each portion of its dynamic range. The resulting distribution will have probability for values in the dynamic ranges of **all** partial accumulations, and so the larger dynamic range will imply high entropy.

B. Computation Reordering Strategies

There are two natural strategies for performing quantized accumulation: a linear accumulation and a binary tree. Each corresponds to a data-flow diagram and consequently to a precedence order for the operations. They are "natural" in that they closely parallel common data structures used for dealing with an array of values, and in fact represent the extremes of a large set of possibilities. Linear accumulation is a straightforward accumulative algorithm, where the next value of a variable equals the current value plus an additional value as depicted on the left in Figure 1. In the case of $e_i(t)$, the accumulator needs to be initialized to zero. $e_i(t) = e_i(t) + d_{i,k}(t)$ is then consecutively performed for $k = 1..D$. An alternative to linear accumulation is to use a binary tree as depicted on the right in Figure 1, where the original inputs are combined separately and the outputs are again combined separately.

There are different ways of implementing these two schemes with LUTs. First, a separate table could be used for each circle, where each table would be customized to its particular distribution of input and output values. This would lead to small tables since each table would be customized for the typical dynamic range of its operands, but $D-1$ distinct tables would be needed.

At the other extreme, a single table could be used for all circles but it must account for enough values to adequately

[1]Log addition in our system was implemented in a more efficient way.

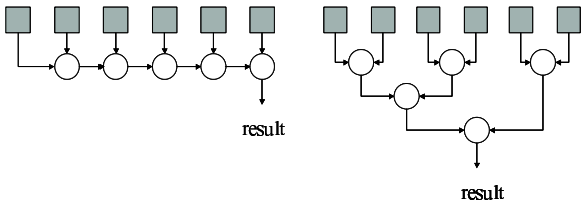

Fig. 1. Linear accumulation vs. tree-structure accumulation. Squares refer to operands; circles refer to arithmetic operators to be implemented by table lookup. Note that a separate LUT could be used for each circle, one large LUT could be used for all circles, or some subset of circles could share from a set of LUTs.

represent the value range produced by the operation. Therefore, the table itself might be very large since the distributions of its inputs and output would have much higher entropy. Using a single table, however, may work well in cases such as $\bar{b}_j(t) = \bigoplus(\cdot)$, where the log addition operator does not dramatically change the value range between inputs and output at each iteration. Similarly, if an operation is iterated only a small number of times, a single table may also be sufficient.

Between these two extremes, fewer than $D-1$ tables can account for all $D-1$ circles. There is a tradeoff between the number of tables and table size — a smaller number of tables requires larger but shared codebooks. One solution in the tree case is to use a separate table for each tree level, leading to $\lceil \log D \rceil$ tables. The potential advantage here is that each tree level can expect to see input operands with a similar dynamic range and output operand distributions with lower entropy. As a result, the total table size may be minimized.

This work compares three strategies in computing $e_i(t)$. First, we implemented the linear case using one shared operator (LUT). Second, we tried two different tree accumulation patterns, both with $\lceil \log D \rceil$ operators as mentioned above. The two tree strategies differ only at the top level. As shown on the left in Figure 2, the first case adds adjacent elements of the vector $\{d_{i,k}(t)\}_{k=1}^{D}$, but at the next tree level half the inputs consist of combined static features and the other half consist of combined deltas. The second case, depicted on the right in Figure 2, instead combines the $d_{i,k}(t)$ of each feature with its corresponding delta. This idea is based on the empirical observation that the dynamic range of $d_{i,k}(t)$ is similar between the static features after mean subtraction and variance normalization, and similar also between the deltas. This will cause the outputs of the top level to be more homogeneous, leading to better quantization and representation. When dealing with an incomplete tree (D is not a power of 2), some values are allowed to pass over levels and are added to lower levels of the tree, as shown on the right in Figure 1.

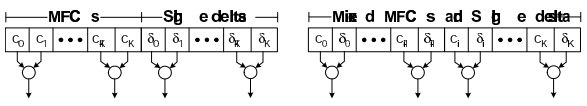

Fig. 2. Adding adjacent elements vs. re-ordering elements to add MFCCs and corresponding deltas; only the first level of the accumulation is shown, with the array elements corresponding to the boxes of Fig. 1.

IV. Normalization of Viterbi Search

Viterbi search is another heavy load for an ASR engine. A key goal of this work is for custom arithmetic to be applicable to Viterbi search as well.

A. Problem Formulation

In decoding, the forward probability $\alpha_j(t) = P(O_{1:t}, q_t = j)$ is calculated as

$$\bar{\alpha}_j(t) = [\sum_i (\bar{\alpha}_i(t-1) + \bar{a}_{ij})] + \bar{b}_j(t) \quad (2)$$

or, using the Viterbi approximation to exact inference,

$$\bar{\phi}_j(t) = [\max_i (\bar{\phi}_i(t-1) + \bar{a}_{ij})] + \bar{b}_j(t) \quad (3)$$

with the final score approximated as $\log P(O_{1:T}) \approx \max_j [\bar{\phi}_j(T) + \bar{a}_{jN}]$. Here we let states 1 and N denote the beginning and ending non-emitting states respectively.

As can be seen in Equation (2) and Equation (3), neither $\bar{\alpha}$ nor $\bar{\phi}$ has a bounded dynamic range. Specifically, as T increases these log Viterbi scores will decrease, causing severe problems in codebook design. Since the utterance length in real applications is unknown at the system design stage, the $\bar{\alpha}$ (or $\bar{\phi}$) values at decode time might not lie in the dynamic range of those values used for quantization at codebook design time. Essentially, the distribution over $\bar{\alpha}$ has high entropy since the values decrease unboundedly with T. While we could assume some upper bound on T and quantize with $\bar{\alpha}$ distributed accordingly, this would yield an exponentially larger and wasteful codebook with many values seldom used by short utterances. Therefore, we need a normalized version of the forward probability, where inference is still valid but the dynamic range is restricted regardless of utterance length.

B. Within-word Normalization

In this subsection we modify the notation by adding a subscript k to distinguish between values associated with different word models W_k, $k = 1..K$. Also, Q_k denotes the set of states in word W_k.

In the literature, $\alpha'_{j,k}(t) \triangleq P(q_t = j | O_{1:t}, W_k)$ has commonly served as a normalized forward probability to solve the underflow problem that occurs in the Baum-Welch algorithm using fixed-precision floating-point representation [21], [22]. This is equivalent to rescaling $\alpha_{j,k}(t)$ by $P(O_{1:t}|W_k)$, producing a quantity with a representable numerical range. The recursion then becomes

$$\alpha'_{j,k}(t) = \frac{P(O_{1:t-1}|W_k)}{P(O_{1:t}|W_k)} \left[\sum_{i \in Q_k} \alpha'_{i,k}(t-1) a_{ij,k} \right] b_{j,k}(t) \quad (4)$$

$$= \frac{1}{R_k(t)} \left[\sum_{i \in Q_k} \alpha'_{i,k}(t-1) a_{ij,k} \right] b_{j,k}(t), \quad (5)$$

where $R_k(t) = P(O_t | O_{1:t-1}, W_k)$ is computed as

$$R_k(t) = \sum_{j \in Q_k} [\sum_{i \in Q_k} \alpha'_{i,k}(t-1) a_{ij,k}] b_{j,k}(t). \quad (6)$$

However, computing $\alpha'_{j,k}(T) = P(q_T = j|O_{1:T}, W_k)$ alone is insufficient to obtain the final likelihood score $P(O_{1:T}|W_k)$ needed in decoding. Obtaining this score requires $\log R_k(t)$ to be stored during the forward pass and be summed up over t at the end, again giving an ever growing dynamic range.

C. Cross-word Normalization

The essential problem with within-word normalization is that the scaling factor is different for different word models at each frame. The final α' scores, therefore, are not comparable to each other. One standard approach [21] to circumvent this problem is to use the same scaling factor at each frame for all word models. Formally, we introduce $\alpha''_{j,k}(t)$ with a recursion

$$\alpha''_{j,k}(t) = \frac{1}{S(t)}[\sum_{i\in Q_k} \alpha''_{i,k}(t-1)a_{ij,k}]b_{j,k}(t). \tag{7}$$

where

$$\begin{aligned} S(t) &= \textstyle\sum_k \sum_{j\in Q_k}[\sum_{i\in Q_k} \alpha''_{i,k}(t-1)a_{ij,k}]b_{j,k}(t) \\ &\approx \max_{k,j\in Q_k}[\textstyle\sum_{i\in Q_k} \alpha''_{i,k}(t-1)a_{ij,k}]b_{j,k}(t). \end{aligned} \tag{8}$$

This scaling factor in α'''s recursion is independent of W_k, and it naturally obtains the score for W_k at the end of the corresponding forward pass:

$$\hat{k} = \operatorname*{argmax}_k \sum_{j\in Q_k} \alpha''_{j,k}(T)a_{jN,k}. \tag{9}$$

There are potential difficulties, however, with implementing this recursion. As can be seen in Equations (7) and (8), computing the scaling factor involves significant additional operations. When implemented with custom arithmetic, there is the additional problem that the total table size might still be large since extra LUTs are needed for the scaling operation.

D. Time-invariant Normalization

Under modest assumptions, we show that the dynamic range of $\bar{\phi}$ is bounded by linear functions of time. Equation (3) suggests that

$$\begin{aligned} \max_j \bar{\phi}_j(t) - \max_i \bar{\phi}_i(t-1) &\leq \max_{ij} \bar{a}_{ij} + \max_j \bar{b}_j(t) \\ \min_j \bar{\phi}_j(t) - \min_i \bar{\phi}_i(t-1) &\geq \min_{ij} \bar{a}_{ij} + \min_j \bar{b}_j(t). \end{aligned} \tag{10}$$

Equation (10) can be written recursively for all frames and summed up on both sides, leading to

$$\begin{aligned} \max_j \bar{\phi}_j(t) &\leq t \cdot \max_{ij} \bar{a}_{ij} + \sum_{s=1}^{t} \max_j \bar{b}_j(s) \\ \min_j \bar{\phi}_j(t) &\geq t \cdot \min_{ij} \bar{a}_{ij} + \sum_{s=1}^{t} \min_j \bar{b}_j(s). \end{aligned} \tag{11}$$

Assuming $\max \bar{b}_j(t)$ is a mean ergodic process, namely $\frac{1}{t}\sum_{s=1}^{t} \max \bar{b}_j(s) = E[\max \bar{b}_j(t)]$, and similarly for the min case, we have

$$r_l t \leq \bar{\phi}_j(t) \leq r_h t \tag{12}$$

where $r_h \triangleq \max_{ij} \bar{a}_{ij} + \mathrm{E}[\max_j \bar{b}_j(t)]$ and $r_l \triangleq \min_{ij} \bar{a}_{ij} + \mathrm{E}[\min_j \bar{b}_j(t)]$.

Motivated by Equation (12), we propose a normalized forward probability $\eta_j(t) \triangleq \phi_j(t)e^{rt}$, where r is a positive constant. The final Viterbi score consequently becomes

$$\max_j[\bar{\eta}_j(T) + \bar{a}_{jN}] \approx \log P(O_{1:T}) + rT \tag{13}$$

First, Equation (13) is a valid scoring criterion because the offset rT stays the same for all word candidates and hence has no impact on the final decision. This allows us to choose the r that best normalizes the forward probability. Second, dynamic programming still applies to the inference:

$$\bar{\eta}_j(t) = \max_i[\bar{\eta}_i(t-1) + \bar{a}_{ij}] + \bar{b}_j(t) + r. \tag{14}$$

As shown in Section VII, this normalization does not require extra table lookup operations. Finally, the dynamic range of the normalized log forward probability $\bar{\eta}$ is controlled by r, since by Equation (12)

$$(r_l + r)t \leq \bar{\eta}_j(t) \leq (r_h + r)t. \tag{15}$$

To choose r, we compute the scores of all utterances from the training set evaluated on their own generative word models, and let $r = -\ \mathrm{E}\ [\log P(O_{1:T}|\text{correct model})/T]$, in an attempt to normalize to zero the correct model's log likelihood score with zero word error. It still might be true that when evaluating utterances on the test set with respect to a wrong word model, or if the right model happens to currently be in error, the score decreases as T increases. When this happens, however, it will be for those words with lower partial likelihoods, and are (hopefully) in error. The scheme may act as pruning away unpromising partial hypotheses by encoding their likelihoods with a very few number of bits.

V. Discriminatively Inspired Distortion Measure

Lacking an analytically well-defined distortion measure to maximize recognition rate, conventional discrete-density HMM based ASR systems often use Euclidean or Mahalanobis distance for VQ [21]. Here, we present a new metric customized to minimize the degradation in recognition accuracy.

As will be shown in Section VIII, the forward probability requires the highest bit-width among all system variables. We are therefore particularly interested in further compressing this variable. Forward probabilities are in fact just likelihoods. The correct answer will typically have a high likelihood, whereas very wrong answers will typically have low likelihoods and are likely to be pruned away. A standard data-driven quantization scheme, however, tends to allocate more bits to a value range based only on its probability mass. Since low likelihoods are more probable (there is only one correct answer and many wrong ones), more bits will be allocated to these low scores at quantization time, thereby giving them excessively high resolution.

Therefore, we propose a discriminatively inspired distortion measure to penalize low-valued forward probabilities. We

define the distortion between a sample $\bar{\eta}_j(t) = x$ and its quantized value $Q(x)$ as

$$D(x, Q(x)) = \frac{(x - Q(x))^2}{f(x)}, \tag{16}$$

where $f(x)$ is strictly positive. In choosing $f(x)$, it is desired that as x increases, the distance between x and $Q(x)$ will increase, which will cause more bits to be allocated for higher likelihood scores. $f(x) = s - x$ is such a function, where $s > \max x$ controls the degree of discrimination with smaller s implying higher discrimination. In this work, s was determined empirically from training data.

VI. Optimization of Bit-Width Allocation

So far, we have discussed table construction, but have not addressed how to determine the size of each table. The goal is to come up with a system-wide optimization algorithm to allocate resources among all variables. We aim to find the bit-width allocation scheme which minimizes the cost of resources while maintaining baseline recognition performance.

The algorithms will be presented for a system with L variables $\{V_i\}_{i=1}^{L}$. We now define the following:

- $\mathbf{fp}$ – The bit-width of an unquantized floating-point value. Typically, $\mathbf{fp} = 32$;
- bw_i – the bit-width of V_i. bw_i can take any integer value below $\mathbf{fp}$. When $bw_i = \mathbf{fp}$, V_i is unquantized;
- $\vec{bw} = (bw_1, bw_2, ..., bw_L)$ – a bit-width allocation scheme;
- $\Delta\vec{bw}_i$ – an increment of 1 bit for variable V_i, where $\vec{bw} + \Delta\vec{bw}_i = (bw_1, ..., bw_i + 1, ..., bw_L)$;
- $wer(\vec{bw})$ – the word error rate (WER) evaluated at $\vec{bw}$;
- $cost(\vec{bw})$ – total cost of resources evaluated at $\vec{bw}$.

Note that the cost function can be arbitrarily defined depending on the specific goals of the allocation. In this paper, we use the total storage of the tables as the cost. Additionally, we define the gradient δ as the ratio of the decrease in WER to the increase in cost evaluated in a certain axis direction, for example,

$$\delta_{ij}(\vec{bw}) \triangleq \frac{wer(\vec{bw}) - wer(\vec{bw} + \Delta\vec{bw}_i + \Delta\vec{bw}_j)}{cost(\vec{bw} + \Delta\vec{bw}_i + \Delta\vec{bw}_j) - cost(\vec{bw})} \tag{17}$$

reflects the rate of improvement along the joint direction of bw_i and bw_j. For a single-dimensional increment, we set $j = 0$: $\delta_i(\vec{bw}) = \delta_{i0}$.

We define a baseline WER $\text{BWER} \triangleq wer(\vec{bw})|_{\vec{bw}=(\mathbf{fp},\mathbf{fp},...,\mathbf{fp})} + \epsilon$, with ϵ a tolerance for increased error due to quantization. Our goal can be interpreted as

$$\vec{bw}^{\,*} = \underset{\vec{bw}: wer(\vec{bw}) \leq \text{BWER}}{\text{argmin}} \; cost(\vec{bw}) \tag{18}$$

This search space is highly discrete and, due to the effects of quantization noise, only approximately smooth. An exhaustive search for $\vec{bw}^{\,*}$, evaluating $wer(\vec{bw})$ and $cost(\vec{bw})$ at every possible $\vec{bw}$, is clearly exponential in L. Even constraining the bit-width of each variable to a restricted range of possible values gives a very large search space. In the following subsections, we present two heuristics that work well experimentally. The basic idea is that we start with a low-cost $\vec{bw}$ with low enough a WER that gradients are not meaningless (they provide no information if bit-widths are too low) and greedily increase the bit-widths of one or a small group of variables until we find an acceptable solution. This is similar to the method used in [23] to optimize floating-point bit-widths.

A. Single-Variable Quantization

Finding a reasonable starting point is an important part of these algorithms. In general, it is expected that the noise introduced into the system by quantizing an additional variable will produce a result that is not better than without the extra variable quantized. For that reason, we take the starting point to be the minimum number of bits needed to quantize a single variable to produce baseline WER results. We call that result m_i, the minimum bit-width of variable V_i. We determine an upper bound M_i by inspection. Specific methods for quantizing individual variables have been introduced in the previous sections.

Once we determine the boundaries for each single variable, we have constrained our search to the hypercube bounded by $m_i \leq \vec{bw}_i \leq M_i$, $i = 1..L$. We then start each of the following algorithms at $\vec{bw}_{init} = (bw_1 = m_1, bw_2 = m_2, ..., bw_L = m_L)$. In all cases, it is assumed that $wer(\vec{bw}_{init}) > \text{BWER}$ since the algorithms are designed to stop when they find a $\vec{bw}$ with a WER as low as the target rate.

B. Single-Dimensional Increment

This algorithm allows only single-dimensional increments. It uses the gradient $\delta_i(\vec{bw})$ as a measure of improvement. The algorithm is described as follows,

1. Evaluate the gradients $\delta_i(\vec{bw})$, $i = 1..L$, for the current $\vec{bw}$ according to Equation (17), where L tests are needed to obtain the WERs. If $\delta_i(\vec{bw}) < 0 \; \forall i$, return $\vec{bw}^{\,*} = \vec{bw}$;
2. Choose the direction $k = \underset{i: bw_i + 1 \leq M_k}{\text{argmax}} \; \delta_i(\vec{bw})$, and set $\vec{bw} = \vec{bw} + \Delta\vec{bw}_k$;
3. If $wer(\vec{bw}) \leq \text{BWER}$, return $\vec{bw}^{\,*} = \vec{bw}$; otherwise repeat steps 1 and 2.

For speech recognition, online evaluation for a test takes a significant amount of time. As this takes place during the design stage, this is not a problem. Note that there might exist the case that no improvement exists along each of the L directions, but that one does exist with joint increments along multiple dimensions. With this algorithm, the search can easily become stuck in a local optimum.

C. Multi-Dimensional Increment

To avoid local optimum, the bit-widths of multiple variables could be increased in parallel. Considering the computational complexity, we only allow one- or two-dimensional increments, leading to $L + \binom{L}{2}$ possible candidates. We could extend this to include triplet increments, but it would take an

intolerably long time to finish. Only steps 1 and 2 differ from the above algorithm, becoming:

1. Evaluate the gradients $\delta_i(\vec{bw})$ and $\delta_{ij}(\vec{bw})$, $i = 1..L$, $j = 1..L$ on current $\vec{bw}$ where $L + \binom{L}{2}$ tests are needed to obtain the WERs. Return $\vec{bw}^* = \vec{bw}$ if all these gradients are negative;
2. Choose the direction k or a pair of directions $\{k, l\}$ where $\delta_k(\vec{bw})$ or $\delta_{kl}(\vec{bw})$ is the maximum among all the single-dimensional and pair-wise increments. Increase the bit-width of V_k or those of $\{V_k, V_l\}$ by one if no one exceeds its upper bound.

This algorithm is superior to the first one in the sense that it explores many more candidate points in the search space. It considers only one additional direction and may still fall into a local optimum, but is less likely to do so than in the previous case. The downside is that this algorithm takes substantially longer to complete.

VII. System Organization

A. Baseline System Configuration

The database used for system evaluation is NYNEX PhoneBook [24], a phonetically-rich speech database designed for isolated-word recognition tasks. It consists of isolated-word utterances recorded via telephone channels with an 8,000 Hz sampling rate. Each sample is encoded into 8 bits according to μ-Law. We set aside 79778 utterances for training, 6598 for development and 7191 for evaluation.[2] The development set is comprised of 8 different subsets, each with a different 75-word vocabulary. For a comprehensive testing, the evaluation set is divided in four ways: a) 8 subsets each with a 75-word vocabulary; b) 4 subsets each with a 150-word vocabulary; c) 2 subsets each with a 300-word vocabulary and d) one set with a 600-word vocabulary. Besides the experiments in clean conditions, artificial white Gaussian noise is added to the evaluation set generating utterances with SNRs of 30dB, 20dB and 10dB.

The acoustic features are the standard MFCCs plus the log energy and their deltas, yielding 26-dimensional vectors. Each vector has mean subtraction and variance normalization applied to both static and dynamic features in an attempt to make the system robust to noise.

The acoustic models are a set of phone-based CHMMs. This enables a customizable vocabulary, a desirable goal for an embedded device. Our system has 42 phone models, each composed of 4 emitting states except for the silence model which has 1 emitting state. The state probability distribution is a mixture of 12 diagonal Gaussians.

The front-end and the back-end are two main components of the recognizer, where the back-end has been discussed in previous sections. Our front-end consists of active speech detection and feature extraction. It expands each μ-law encoded sample into linear 16-bit PCM, and then creates a frame every 10ms (80 samples), each with a length of 25ms (200 samples). The speech detector used is one of the simplest; it detects speech when the energy level of the speech signal rises above a certain threshold. This design uses minimal extra resources while still accurately detecting speech when noise conditions are stationary. Feature extraction is triggered immediately when active speech is detected. It follows a standard procedure described in [25], We then add the first order dynamic features followed by mean subtraction and variance normalization. The feature vectors obtained are fed into the back-end, where the pattern matching takes place. Since we propose applying our custom arithmetic to the back-end but not to the front-end, an interface is necessary. At the interface, the floating-point value of a feature element is converted into its integer index in the associated codebook by a binary search. This is in fact the only place in the system where a software codebook search is needed, and is the biggest overhead introduced by custom arithmetic. This overhead is taken into account in our CPU time simulations.

B. Codebook and Table Definition

Based on the analysis in the previous sections, we defined 13 variables to be quantized which are listed in Table I. The variable $\mathbf{x}$ is the output feature element of the front-end, $\mathbf{m}, \mathbf{v}, \mathbf{c}, \mathbf{w}$ and $\mathbf{a}$ are the acoustic model parameters pre-computed, and $\mathbf{s}, \mathbf{d}, \mathbf{e}, \mathbf{p}, \mathbf{q}, \mathbf{b}$ and η are other intermediate variables in the back-end system.

TABLE I
SYSTEM VARIABLES AND THEIR CORRESPONDING EXPRESSIONS

symbol	in Equation (1)
$\mathbf{x}$	$x_k(t)$
$\mathbf{m}$	$\mu_{i,k}$
$\mathbf{v}$	$\sigma^2_{i,k}$
$\mathbf{c}$	c_i
$\mathbf{w}$	$\bar{w}_i$

symbol	in Equation (1)
$\mathbf{s}$	$(x_k(t) - \mu_{i,k})^2$
$\mathbf{d}$	$d_{i,k}(t)$
$\mathbf{e}$	$e_i(t)$
$\mathbf{p}$	$c_i - \frac{1}{2}e_i(t)$
$\mathbf{q}$	$\bar{w}_i - c_i - \frac{1}{2}e_i(t)$
$\mathbf{b}$	$\bar{b}_j(t)$
$\mathbf{a}$	$\bar{a}_{i,j}$
η	$\bar{\eta}_j(t); \bar{\eta}_j(t) + \bar{a}_{ij}$

There are 8 functions and hence 8 potential LUTs associated with these variables:

$$F_1(\cdot): \mathbf{s} = (\mathbf{x}-\mathbf{m})^2 \qquad F_5(\cdot): \mathbf{q} = \mathbf{w}+\mathbf{p}$$
$$F_2(\cdot): \mathbf{d} = \mathbf{s}/\mathbf{v} \qquad F_6(\cdot): \mathbf{b} = \mathbf{b} \oplus \mathbf{q}$$
$$F_3(\cdot): \mathbf{e} = \mathbf{e}+\mathbf{d} \qquad F_7(\cdot): \eta = \eta + \mathbf{a}$$
$$F_4(\cdot): \mathbf{p} = \mathbf{c}-\mathbf{e}/\mathbf{2} \qquad F_8(\cdot): \eta = \eta + \mathbf{b} + \mathbf{r}$$

$F_1(\cdot)$ and $F_2(\cdot)$ involve floating-point multiplication and division respectively, and these operations would be performed millions of times for an ordinary isolated word recognition task. $F_6(\cdot)$ would be executed thousands of times with even more expensive log computation. We expect the simple hardware-based lookup operations will dramatically save cycles as well as power on these functions.

Note that the comparison operations implicit in Equation (14) are easily implemented, since they can be achieved using low bit-width integer comparisons operations, so no extra tables are required.

[2]Specifically, the training set consisted of subsets aa, ab, ah, ai, am, an, aq, at, au, ax, ba, bb, bh, bi, bm, bn, bq, bt, bu, bx, ca, cb, ch, ci, cm, cn, cq, ct, cu, cx, da, db, dh, di, dm, dn, dq, dt, du, dx, ea and eb. The development set included subsets ad, ar, bd, br, cd, cr, dd and dr. Finally, the evaluation set was comprised of subsets ao, ay, bo, by, co, cc, do, dy.

VIII. Experiments and Results

Before reporting our experiments and results, we would like to clarify two points. First, the recognition program used in all our simulations does not utilize purely algorithmic methodology such as Gaussian selection, beam pruning or algorithmic-level vector quantization techniques; incorporating these procedures may indeed affect (and likely improve) our speedup factors. But in this work we concentrate solely on architectural customization and enhancement. Second, we must acknowledge that an alternative architectural strategy is to utilize a "hybrid" system consisting of a mix between floating- and fixed-point arithmetic operations. Our contention, however, is that when the greatest possible power savings must be obtained, custom designed arithmetic operations (as we employ in this work) will yield the best tradeoff between ASR accuracy and power consumption reduction.

This section first reports the results of system development. The LBG [26] algorithm was used in all single variable quantization experiments. These experiments were performed on the 8 development subsets mentioned in the previous section. The WERs reported were an average over these subsets. We also tried 150-, 300- and 600-word vocabulary cases on the development set and observed similar trends. Based on the single-variable quantization experiments, we applied our search algorithms for resource allocation, where the WERs were again evaluated on the development set. Next we evaluated the recognition accuracy of the best allocation scheme in both clean and noisy conditions on our evaluation set. Finally, we estimated its memory usage and simulated its speed performance.

A. System development

In the single-variable quantization experiments, we quantize each variable individually, leaving all other variables at full precision. The baseline WER of the development set is 2.07%.

Table II shows the minimum bit-width to which a variable can be quantized without any increase in WER on the development set. Here we report all system variables except for the accumulated Mahalanobis distance $\mathbf{e}$ and the forward probability η which will be discussed separately later. Note that we let $\mathbf{q}$ and $\mathbf{b}$ share the same codebook because their value ranges have much overlap.

TABLE II
MINIMUM BIT-WIDTH TO WHICH EACH VARIABLE CAN BE INDIVIDUALLY QUANTIZED WITHOUT INCREASE IN WER

variable	$\mathbf{m}$	$\mathbf{v}$	$\mathbf{w}$	$\mathbf{a}$	$\mathbf{x}$	$\mathbf{s}$	$\mathbf{d}$	$\mathbf{c}$	$\mathbf{p}$	$\mathbf{b}$ ($\mathbf{q}$)
min bit	4	5	2	2	5	6	5	5	6	5

Section III presented two approaches to quantize variable $\mathbf{e}$. In our case, the operation $\mathbf{e} = \mathbf{e} + \mathbf{d}$ is repeated 26 times to get the final value of $\mathbf{e}$. Using linear accumulation, it involves only two variables $\mathbf{e}$ and $\mathbf{d}$ and only one table $\mathbf{e} = \mathbf{e} + \mathbf{d}$. Alternatively, we can use tree-structure accumulation with multiple variables and $\lceil \log 26 \rceil = 5$ different tables each of relatively small size. For simplicity, the variables in each level of the tree structure are quantized with the same bit-width. Figure 3 summarizes the results in terms of WER vs. bit-width of each codebook. Tree-structure 1 denotes the method where adjacent pairs in the vector are added at all levels to form the next-level codebook, whereas in tree-structure 2, MFCCs are added to their corresponding deltas at the first level. It can be seen that in order to achieve the baseline recognition rate, linear accumulation needs 7 bits and tree-structure schemes need 6 bits for each codebook. The total table size, however, is a different story since the tree-structure schemes require 5 separate LUTs. To compute the total table size, we only consider the two functions in which $\mathbf{e}$ is involved.[3] Assuming $n_d = 5$, $n_c = 5$ and $n_p = 6$ according to Table II, 6 kBytes of LUTs are needed to realize the operations involving $\mathbf{e}$ using linear accumulation, whereas the required space goes up to almost 10 kBytes for the tree-structure schemes to achieve the same goal. It is worth noting that the feature dimension is fixed at 26 and is relatively low, and the addition operation only changes the dynamic range of the output at a linear scale. This yields only a mild increase in the entropy of $\mathbf{e}$, thereby making the linear accumulation an effective approach. Nevertheless, the tree structure may show advantages for other types of operations such as a long sequence of multiplications.

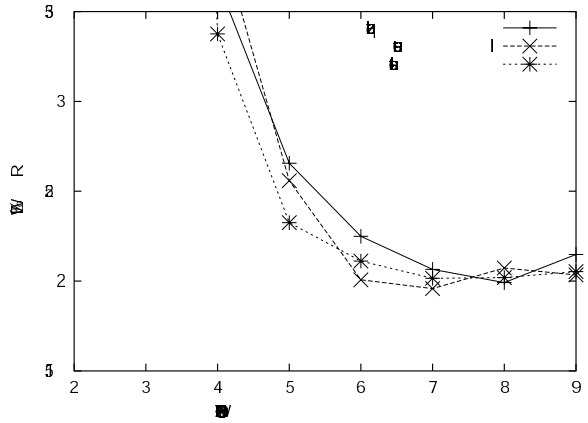

Fig. 3. Single-variable quantization for accumulative variable $\mathbf{e}$

Section IV proposed a time-invariant normalization to the forward probability to reduce its entropy without affecting recognition decision. To show the advantage of the normalization on quantization, we extracted forward probability samples with and without normalization on the same subset of training data, and generated codebooks based on the Euclidean distance distortion measure for each case. We additionally applied quantization of the normalized forward probabilities using our discriminatively inspired distortion measure. As shown in Figure 4, the normalized Viterbi search obviously outperforms the unnormalized case by saving 1 bit while keeping the baseline recognition rate (thus nearly halving the total table size). In fact, we believe the benefits of normalization would be more pronounced on a task with longer utterances, such as connected-digit or continuous speech recognition. In addition,

[3]With $\mathbf{e}$, $\mathbf{d}$, $\mathbf{c}$ and $\mathbf{p}$ quantized to n_e, n_d, n_c and n_p bits respectively, all related tables total $n_e 2^{n_d+n_e} + n_p 2^{n_c+n_e}$ bits.

the discriminative distortion measure works slightly better than the normal one.

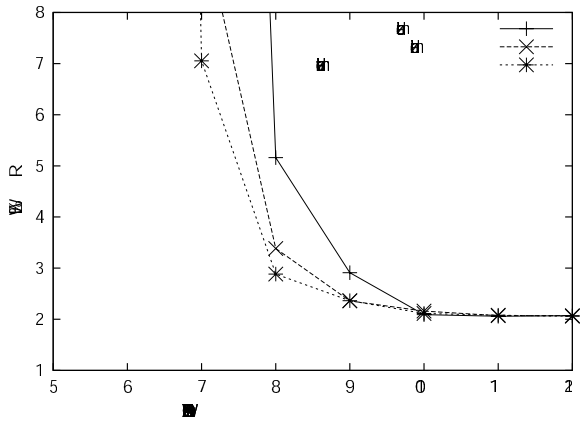

Fig. 4. Single-variable quantization for forward probability

We therefore chose linear accumulation in quantizing **e** and normalized Viterbi search using our distortion measure in quantizing the forward probability. Together with other variable quantization results, codebooks with different resolution were generated for all system variables, to which an optimization search was applied to find the best bit-width allocation scheme. The results of running both allocation algorithms appear in Table III. Recall that the baseline WER of the development set is 2.07%.

TABLE III
COMPARISON OF TWO BIT-WIDTH OPTIMIZATION ALGORITHMS

	WER	Table (kB)	# Tests
1-D	3.09%	36.44	52
2-D	2.53%	58.75	454

The 1-D (single-dimensional increment) algorithm tested a total of 52 configurations before it reached a local optimum. It managed to find a fairly small ROM size but the WER was not very satisfactory. The 2-D (two-dimensional increment) algorithm found a much better WER than 1-D, even if its total table size was larger.

The final bit-width allocation scheme of the last algorithm is shown in Table IV, where the first row indicates the variable and the second row shows the corresponding bit-width. As

TABLE IV
THE OPTIMAL BIT-WIDTH ALLOCATION SCHEME

V	**m**	**v**	**w**	**x**	**s**	**d**	**e**	**c**	**p**	**b**	**a**	η
BW	6	7	3	7	7	6	7	5	8	5	3	9

shown in the table, all the variables can be compressed to less than 10 bits, which substantially reduces the memory band-width. This scheme takes only 59 kBytes of memory for table storage, an amount affordable for most modern chips. It is also interesting to see that the responsibility **w**, transition probability **a**, mean scalar **m**, variance scalar **v** and constant **c** can each be quantized to less than 8 bits, leading to an 80% reduction of model parameter storage as opposed to 32-bit floating-point representation. In addition, the feature scalar **x** and the state likelihood **b** can be quantized to 7 and 5 bits respectively, resulting in an additional saving in online memory usage.

For some computer architectures, 8- or 16-bit is the bit-width that can be most effectively used. For example, the design of a lookup instruction would be easier if its operands are uniformly 8 bits (those smaller can be filled with zeros at the beginning). For this reason, we additionally conducted an optimization experiment with the constraint that all variables were quantized to no more than 8 bits. Note that to minimize the table size, variables with fewer than 8 bits were not expanded to 8 bits. The resulting bit-width allocation scheme uses 68.5 kB to achieve a WER of 3.22%. We would also like to mention that for custom chips and reconfigurable logic chips, 8- or 16-bit is not always necessary since buses and registers can have smaller bit-widths.

B. Final system evaluation

We tested the recognition performance of the scheme in Table IV on our evaluation set for all 75-, 150-, 300- and 600-word vocabulary cases as defined in Section VII. The experiments were done in both clean and noisy conditions. We did not apply any noise-robustness techniques except for mean subtraction and variance normalization to the MFCC and delta features.

As shown in Table V, for recognition in relatively clean conditions (clean and SNR=30dB cases), the system using custom arithmetic units has slight degradation in recognition rate compared to the baseline system using a floating point unit. The maximum degradation, an absolute 1.7% increase in WER, happens in the 600-word and 30dB case. This is graceful considering the potential speedup that custom arithmetic brings, which is discussed in the next subsection. It is interesting to see that in more noisy conditions (SNR=20dB and SNR=10dB cases), custom arithmetic does not deteriorate the recognition performance any more, but on the contrary, slightly enhances it. One explanation is that the quantization noise introduced may, to some extent, compensate for the more continuous additive noise in the speech; so quantization then acts as a regularizer — the decision boundaries are probably slightly less susceptible to minor perturbations when variables are so coarsely quantized in the custom arithmetic case.

TABLE V
%WERS ON EVAL SET USING THE SCHEME IN TABLE IV

vocab.	75-word		150-word		300-word		600-word	
	fp.	custom	fp.	custom	fp.	custom	fp.	custom
clean	2.26	2.99	3.23	4.43	5.09	6.70	19.09	20.90
30 dB	2.68	3.46	3.91	4.98	6.26	7.58	20.62	22.33
20 dB	8.55	8.23	11.71	11.13	16.03	15.40	31.14	31.11
10 dB	30.65	29.59	38.36	36.87	46.21	44.44	58.78	57.71

C. CPU Time Simulation

We utilized SimpleScalar [27], an architecture-level execution-driven simulator, for CPU time simulation. All

tables were pre-calculated by SimpleScalar at runtime. We extended the instruction set and modified the assembler to support our new lookup instructions.

The simulation was targeted for a variety of architectures. Without a detailed manufacturer-supplied simulator, an in-depth analysis of any specific architecture is impossible, but by using a range of simulated architectures we can still discover meaningful trends in performance. The first is the SimpleScalar default configuration, roughly akin to a third generation Alpha, an out-of-order superscalar processor. The second represents a more modern desktop machine; cache sizes and latencies have been updated from the defaults, several years old, to reflect more current values. Our third configuration attempts to represent a typical high-end DSP chip. Since most DSP chips are Very Long Instruction Word (VLIW) architectures, a feature not replicable with SimpleScalar, we chose instead to force in-order execution with high parallelism as a very rough approximation of VLIW. Finally, we simulated a very simple processor with a single pipeline and no cache. This last case is similar to what could be expected of a low-power processor designed to run on an extremely portable device, the expected target platform for custom arithmetic. In each case, we allowed multiple tables to be accessed in parallel, although a specific table could be used by only one instruction at a time.

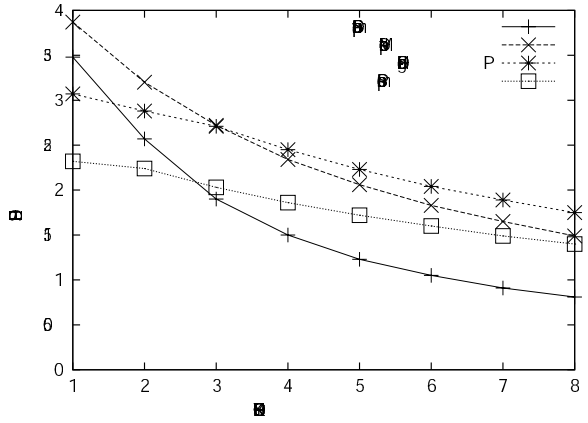

Fig. 5. Speedup (in cycles) versus cycles per lookup

We simulated each of the target machines for a range of table lookup speeds. This is independent of WER since, assuming the table is not too large, the precision of the output of a lookup has little to do with the speed of the lookup. Figure 5 shows the speedup obtained versus the number of cycles required for each lookup. The speedup ranges from just under 2.5 to nearly 4 when lookups take only 1 cycle, and falls off as they become more expensive. If the ROM tables are small enough to fit on the processor chip, which is the case using the 59kB results from Table IV, a 1-cycle lookup is quite realistic for a state-of-the-art system. To be more explicit, we assume that the ROMs will be on-die and implemented using either reconfigurable logic or a custom die for embedded applications. Although this may seem to require a large chip area, there is substantial savings in the on-chip cache size resulting from the reduction in the bit-widths of all variables. As shown in the figure, using a single pipeline and no cache (simple processor), meaning essentially a cache miss every time, does reduce the speedup but still provides a significant gain in speed. Much benefit actually comes from replacing sequences of instructions with a single lookup, as the dynamic instruction count falls nearly as much as execution time. Note that because we did not try to implement a low-power front-end, a feature that would be necessary for a final realization of this system, we used pre-calculated MFCCs in floating-point values and included the time for MFCC quantization when calculating speedups.

IX. Summary and Conclusion

This paper presented a methodology for the design of high-speed, low-resource systems using custom arithmetic units. We focused our attention on the scalar quantization of system variables with high entropy, involving reordering and rescaling of the decoding algorithms and a discriminatively inspired distortion measure. We also demonstrated several resource allocation search heuristics suitable for finding acceptable points in an otherwise intractable search space. Our findings were then applied to a CHMM based ASR system, where a fully-functioning ASR back-end was achieved by LUTs without floating-point arithmetic units. The 59kB of tables is small enough that it can be added to any chip with an access time of 1 cycle. When implementing this design on a modern processor, we show that the expected speedup is at least 3, and possibly larger. Furthermore, the memory required for parameter storage and online computation can be greatly reduced. In addition, we are looking forward to hardware support for our custom arithmetic; the amount of savings in cycles and power also depends on the physical realization of the LUTs and the ISA designed to support lookup operations.

Acknowledgments

We would like to thank Carl Ebeling for useful advice.

References

[1] S. Takahashi, K. Aikawa, and S. Sagayama, "Discrete mixture HMM," in *Proc. Int. Conf. on Acousitics, Speech and Signal Processing*, 1997, vol. 2, pp. 971–974.

[2] V. Digalakis, S. Tsakalidis, C. Harizakis, and L. Neumeyer, "Efficient speech recognition using subvector quantization and discrete-mixture HMMs," *Computer Speech and Language*, vol. 14, pp. 33–46, 2000.

[3] E. Bocchieri, "Vector quantization for the efficient computation of continuous density likelihoods," in *Proc. Int. Conf. on Acousitics, Speech and Signal Processing*, 1993, vol. 2, pp. 692–695.

[4] M. J. F. Gales, K. M. Knill, and S. J. Young, "State based Gaussian selection in large vocabulary continuous speech recognition using HMMs," *IEEE Trans. on Speech and Audio Processing*, vol. 7, 1999.

[5] M. Ravishankar, R. Bisiani, and E. Thayer, "Sub-vector clustering to improve memory and speed performance of acoustic likelihood computation," in *Proc. Eurospeech '97*, 1997, pp. 151–154.

[6] M. Vasilache, "Speech recognition using hmms with quantized parameters," in *Proc. Int. Conf. on Acousitics, Speech and Signal Processing*, 1999.

[7] E. Bocchieri and B. K. Mak, "Subspace distribution clustering hidden Markov model," *IEEE Trans. on Speech and Audio Processing*, vol. 9, no. 3, pp. 264–275, 2001.

[8] K. Filali, X. Li, and J. Bilmes, "Data-driven vector clustering for low-memory footprint asr," in *Proc. Int. Conf. on Spoken Language Processing*, 2002.

[9] K. Filali, X. Li, and J. Bilmes, "Algorithms for data-driven asr parameter quantization," *(accepted) Computer, Speech and Language*.
[10] C. Iordache and P. T. P. Tang, "An overview of floating-point support and math library on the intel xscale architecture," in *16th IEEE Symposium on Computer Arithmetic*, June 2003.
[11] Y. Gong and U. H. Kao, "Implementing a high accuracy speaker-independent continuous speech recognizer on a fixed DSP," in *Proc. Int. Conf. on Acousitics, Speech and Signal Processing*, 2000, pp. 3686–3689.
[12] E. Cornu, N. Destrez, A. Dufaux, H. Sheikhzadeh, and R. Brennan, "An ultra low power, ultra miniature voice command system based on hidden markov models," in *Proc. Int. Conf. on Acousitics, Speech and Signal Processing*, 2002, vol. 4, pp. 3800–3803.
[13] I. Varga and et. al., "ASR in mobile phones – An industrial approach," *IEEE Trans. on Speech and Audio Processing*, vol. 10, no. 8, pp. 562–569, 2002.
[14] B. Mathew, A. Davis, and Z. Fang, "A low-power accelerator for the SPHINX 3 speech recognition system," in *Proc. Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, 2003.
[15] D. Citron, D. Feitelson, and L. Rudolph, "Accelerating multi-media processing by implementing memoing in multiplication and division units," in *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 252–261.
[16] X. Li, J. Malkin, and J. Bilmes, "Codebook design for ASR systems using custom arithmetic units," in *Proc. Int. Conf. on Acousitics, Speech and Signal Processing*, May 2004.
[17] J. Malkin, X. Li, and J. Bilmes, "Custom arithmetic for high-speed, low-resource ASR systems," in *Proc. Int. Conf. on Acousitics, Speech and Signal Processing*, May 2004.
[18] B. Delaney, N. Jayant, M. Hans, T. Simunic, and A. Acquaviva, "A low-power, fixed-point, front-end feature extraction for a distributed speech recognition system," in *Proc. Int. Conf. on Acousitics, Speech and Signal Processing*, 2002, vol. 1, pp. 793–796.
[19] S. J. Melnikoff, P. B. James-Roxby, S. F. Quigley, and M. J. Russel, "Reconfigurable computing for speech recognition: Preliminary findings," in *Proc. Intl. Conf. on Field Programmable Logic and Applications*, 2000, pp. 495–504.
[20] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The chimaera reconfigurable functional unit," *IEEE Trans. on VLSI Systems*, vol. 12, no. 2, pp. 206–217, February 2004.
[21] K-F. Lee, *Automatic speech recognition: the development of the SPHINX system*, Kluwer Academic Publishers, 1989.
[22] J. R. Deller, J. H. Hansen, and J. G. Proakis, *Discrete-time processing of Speech Signals*, Wiley, 1993.
[23] F. Fang, T. Chen, and R. A. Rutenbar, "Floating-point bit-width optimization for low-power signal processing applications," in *Proc. Int. Conf. on Acousitics, Speech and Signal Processing*, 2002, pp. 3208–3211.
[24] J. F. Pitrelli, C. Fong, and H. C. Leung, "PhoneBook: A phonetically-rich isolated-word telephone-speech database," in *Proc. Int. Conf. on Acousitics, Speech and Signal Processing*, 1995.
[25] S. Young, G. Evermann, D. Kershaw, G. Moore, J. Odell, D. Ollason, V. Valtchev, and P. Woodland, *The HTK Book (for HTK Version 3.1)*, Microsoft Corporation and Cambridge University Engineering Dept, 2001.
[26] Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantizer," *IEEE Trans. on Communication*, vol. 28, pp. 84–95, 1980.
[27] D. Burger, T. M. Austin, and S. Bennett, "Evaluating future microprocessors: The simplescalar tool set," Tech. Rep. CS-TR-1996-1308, Dept. of Computer Science, Univ. of Wisconsin-Madison, 1996.
[28] P. Beyerlein and M. Ullrich, "Hamming distance approximation for a fast log-likelihood computation for mixture densities," in *Proc. Eur. Conf. Speech Communication Technology*, 1995, vol. 2, pp. 1083–1086.
[29] J. R. Bellegarda and D. Nahamoo, "Tied mixture continuous parameter modeling for speech recognition," *IEEE Trans. on Acoustics, Speech and Signal Processing*, vol. 38, pp. 2033–2045, 1990.
[30] J. A. Bilmes, "Buried Markov models for speech recognition," in *Proc. Int. Conf. on Acousitics, Speech and Signal Processing*, 1999.

Xiao Li received the B.S.E.E degree from Tsinghua University, Beijing, China, in 2001 and the M.S.E.E. degree from the University of Washington, Seattle, in 2003. She is currently pursuing the Ph.D. degree in electrical engineering at the University of Washington, working on learning and adaptation in voice driven human-computer interaction. Her research interests include statistical learning, acoustic and speech processing and low-resource speech recognition for embedded systems. She is a 2005 recipient of Microsoft research fellowship.

Jon Malkin received the B.S. degree in electrical engineering and computer science from Yale Universitty and the M.S. degree in electrical engineering from the University of Washington in 2002 and 2005, respectively. He is continuing at the University of Washington in pursuing the Ph.D. degree, focusing on vocal parameters and human factors for voice-based human-computer interaction. His research interests include pattern recognition, low-resource computation for real-time systems, speech and audio processing and recognition and human-computer interaction.

Jeff A. Bilmes is an Assistant Professor (Associate, September '05) in the Department of Electrical Engineering at the University of Washington, Seattle (adjunct in Linguistics and also in Computer Science and Engineering). He co-founded the Signal, Speech, and Language Interpretation Laboratory at the University. He received a master's degree from MIT, and a Ph.D. in Computer Science at the University of California, Berkeley. Jeff has done much research on both structure learning of and fast probabilistic inference in dynamic Graphical models. His main research lies in statistical graphical models, speech, language and time series, human-computer interaction, machine learning, and high-performance computing. He was a general co-chair for IEEE Automatic Speech Recognition and Understanding 2003, is a member of IEEE, ACM, and ACL, is a 2001 CRA Digital-Government Research Fellow, and is a 2001 recipient of the NSF CAREER award.