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

Xiao Li, Jonathan Malkin, Jeff Bilmes
{lixiao, jsm, bilmes}@ee.washington.edu
Dept of EE, University of Washington
Seattle WA, 98195-2500
A High-speed, Low-Resource ASR Back-end Based on Custom Arithmetic

Xiao Li, Jonathan Malkin, Jeff Bilmes

{lixiao, jsm, bilmes}@ee.washington.edu
Dept of EE, University of Washington
Seattle WA, 98195-2500

UWEEETR-2004-0019
June 2004

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 a linear/tree-structure accumulation 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 to minimize degradation in 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%.

1 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. Ideally, one could implement a fully-functioning ASR system on a 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.

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 [5–7].
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 compared with a floating-point unit [8]. 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 and power consumption can be significantly reduced using a fixed-point implementation [9–11]. But in a power-constrained system, memory use is critical, and fixed-point arithmetic cuts the available dynamic range without having its representation precision fully utilized. Additionally, some operations can still take numerous processor cycles to complete.

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. For some applications, though, such decisions may be warranted [12]. Many signal processing applications produce system variables (system inputs, outputs and all intermediate values computed within a system) 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. [13] 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, which is extremely fast.

Motivated by empirically observed low entropies of the variables within several speech recognition systems and the efficiency of hardware-based lookup operations, we present a low-resource 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 entropies low enough for low-precision quantization. This involves efforts to modify the decoding algorithms to ensure low entropy for all variables. 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 of the system. While in [14] and [15] 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 results of system evaluation 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 free variables than the front-end since many operations in the back-end 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 (this will be explained more fully in Section 6). In addition, the fixed-point arithmetic for the front-end feature extraction has been well studied and can be implemented by DSPs very efficiently [16]. Additionally, as our results show, we can substantially reduce the necessary number of bits for each variable in the back-end below even that used by standard fixed-point DSPs. The reduction in required bus-width directly provides a significant savings in both power and chip area, as well as indirect savings through the reduction in memory needed for offline parameters. Therefore, we envision a chip that has 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 2 discusses the general mechanism of custom arithmetic and its implementation issues for speech recognition. Section 3 and Section 4 present our computation reordering technique for Gaussian evaluation and our normalization method for Viterbi search respectively. Section 5 formulates a discriminatively inspired distortion measure for scalar quantization. Section 6 investigates several heuristics for bit-width allocation. Section 7 describes our system organization, and Section 8 reports our word error rate (WER) and cycle time experiments and results, followed by concluding remarks.

2 General Design Methodology

In this section we present an ASR system driven by custom arithmetic, where all floating-point calculations are pre-computed and transparent to the online application in the sense that the application can simply treat the calculations
normally without worrying about their implementation. In addition, we address several potential issues associated with custom arithmetic design for an ASR system.

2.1 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,

\[
V_h = F_k(V_i, V_j),
\]

(1)

where \( i, j, h \in \{1..L\} \). \( F_k(\cdot) \), \( k \in \{1..K\} \), can be an arbitrary arithmetic operation or sequence of operations. For example, \( V_h = (V_i - V_j)^2 \) is such a two-operand function.

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 \( I_{V_l}(x) \). A codebook indexed with \( n \) bits can hold \( 2^n \) codewords.

Once the codeword values of the inputs and the output are known, 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

\[
I_{V_h}(z) = T_{F_k}(I_{V_l}(x), I_{V_j}(y))
\]

(2)

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 directly 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, although it is crucial to the cost and power consumption of an embedded system. The implementation of custom arithmetic units can be simplified using reconfigurable logic [17, 18], although at the expense of increased power consumption. Despite the slightly increased design cost, the most likely market seems to be low-power embedded and custom chip devices, such as speech recognition on a watch or a pendant, or for integrating systems into retinal projection devices.

2.2 Design Issues for ASR

In spite of the attractiveness of custom arithmetic, such a system becomes unrealistic if the storage of the tables gets too large. This poses a number of challenges for 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 8, can be quantized to low precision without loss of recognition accuracy. However there do exist several variables with high entropies, 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. Quantizing this accumulative variable using a single codebook may lead to a huge codebook and hence a prohibitive table size. In addition, the forward probability \( \alpha \) in decoding possesses a potentially more fatal problem — the forward pass computes over an arbitrarily long utterance in real applications, making \( \alpha \)’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 3 and Section 4.

The second issue is in fact a quantization problem. Since the bit-width of each variable directly influences the table size, we hope that each variable can 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 $\alpha$, which has the highest entropy even after rescaling, as will be shown in Section 8. Section 5 presents a discriminatively inspired distortion measure to quantize this variable instead of the conventional Euclidean distance metric.

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 6 to find the best search scheme within the existing computational capacity.

3 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 shown 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.

3.1 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, $\bar{\bar{x}} = \log x$. Also, $\oplus$ denotes log addition where $\bar{\bar{x}} \oplus \bar{\bar{y}} = \log(e^\bar{x} + e^\bar{y})$\(^1\). To this end, the log state likelihood $\bar{\bar{h}}_j(t)$ 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

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

(3)

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

As mentioned earlier, many operations in Gaussian evaluation are repetitive. For example, to evaluate the observation probabilities for an utterance with $T$ frames in a system with $M$ different Gaussian components, the operation of the form $(x_k(t) - \mu_{i,k})^2/\sigma^2_{i,k}$ will be performed $T \times M \times D$ times, which probably implies millions of floating-point multiplications. Similarly, the more expensive log addition may be performed thousands of times. It is beneficial, therefore, to substitute simple LUTs for all the operations in the Gaussian evaluation.

A crucial problem inherent to Gaussian likelihood evaluation is that there are two iterative operations suggested by (3). One is $c_i(t) \Delta \sum_{k=1}^{D} d_{i,k}(t)$, where $d_{i,k}(t) \Delta (x_k(t) - \mu_{i,k})^2/\sigma^2_{i,k}$, and the other is $\bar{\bar{h}}_j(t)$ 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 make the assumption 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 of a random variable and the dynamic range of that random variable. The two concepts are related in accumulation. If all input values to be summed are identically distributed, an accumulation with a large dynamic range (relative to its input values) that reaches a value in the upper end of that dynamic range will have seen many partial accumulation values in the low

\(^{1}\)Log addition in our system was implemented in a more efficient way.
and middle portions of its range. This means that taking all the partial accumulation values together, a large dynamic range will also imply a high entropy.

### 3.2 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. We call these two “natural” since they closely parallel structures commonly found in data structures used for dealing with an array of values, and in fact represent the extremes of a large set of possible structures. 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 variable 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.

![Figure 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.](image)

There are different ways of implementing these two schemes with LUTs. First, a separate table could be used for each circle, with the advantage that 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 in both linear and tree cases.

At the other extreme, a single table could be used for all circles but it must account for enough values to adequately 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 entropies. As mentioned in the example at the end of the first subsection, adding $K$ identically distributed elements produces an output with a potential dynamic range that is $K$ times the original dynamic range. Using a single table, however, may work well in cases such as $b_j(t) = \oplus(\cdot)$, 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. The ideal case would use the least amount of overall table size while maintaining the recognition rate, but clearly 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, however, 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 only one overall 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 will 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.

![Figure 1](image)

**Figure 1**: 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 Figure 1.

### 4 Normalization of Viterbi Search

Viterbi search is another heavy load for an ASR engine because the forward probability is calculated for each state at each frame. A key goal of this work is for custom arithmetic to be applicable to Viterbi search as well.

#### 4.1 Problem Formulation

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

$$\bar{\alpha}_j(t) = \left[ \sum_i (\bar{\alpha}_i(t - 1) + \bar{a}_{ij}) \right] + \bar{b}_j(t) \quad (4)$$

or, using the Viterbi approximation to exact inference,

$$\bar{\phi}_j(t) = \left[ \max_i (\bar{\phi}_i(t - 1) + \bar{a}_{ij}) \right] + \bar{b}_j(t) \quad (5)$$

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 (4), the $\alpha$ or $\phi$ scores do not have a bounded dynamic range. Specifically, as $T$ increases the Viterbi score will decrease, something that causes severe problems in both codebook design and fixed-point implementations, as well. First, since the utterance length in real applications is unknown at the stage of system design, the $\alpha$ values at decode time might not lie in the dynamic range of those values used for quantization at codebook design time. Essentially, it is known that the distribution over $\alpha$ has high entropy since the values decrease unboundedly with $T$. While we could assume some upper bound on $T$ and quantize with $\alpha$ distributed accordingly, this would yield an exponentially larger and wasteful codebook with many values seldom used by short utterances. Therefore, it is highly desirable to have a normalized version of the forward probability, where inference is still valid but the dynamic range is restricted regardless of the utterance length.

#### 4.2 Within-word Normalization

In this subsection we modify the notation by adding a subscript $k$ to distinguish the forward probabilities, the transition probabilities, the observation probabilities and the likelihood scores of different word models $W_k, k = 1\ldots K$. Also, we let $Q_k$ denote the set of states within word $W_k$.

In the literature, $\alpha'_{j,k}(t) \overset{\Delta}{=} P(q_t = j | O_{1:t}, W_k)$ has served as a normalized forward probability to solve the underflow problem that occurs in the Baum-Welch algorithm using fixed-precision floating-point representation [19, 20]. This is equivalent to rescaling $\alpha_{1,k}(t)$ by $P(O_{1:t} | W_k)$, producing a quantity with a representable numerical range.
The recursion then becomes

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

(6)

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

(7)

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

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

(8)

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, which again brings up the issue of an ever growing dynamic range.

### 4.3 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 $\alpha'$ scores, therefore, are not comparable to each other. One standard approach [19] 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

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

(9)

where

$$
S(t) = \sum_{k} \sum_{j \in Q_k} \left[ \sum_{i \in Q_k} \alpha''_{i,k}(t-1)\alpha_{ij,k} \right] b_{j,k}(t). 
$$

(10)

or, approximately,

$$
S(t) \approx \max_{k,j \in Q_k} \left[ \sum_{i \in Q_k} \alpha''_{i,k}(t-1)\alpha_{ij,k} \right] b_{j,k}(t). 
$$

(11)

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

$$
\hat{k} = \arg \max_k \sum_{j \in Q_k} \alpha''_{j,k}(T) a_{jN,k}. 
$$

(12)

There are potential difficulties, however, with implementing this recursion. As can be seen in Equations (9), (10) and (11), computing the scaling factor involves additional operations. Although this normalization method reduces the entropy of the forward probability, this is at the cost of CPU time and power. 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.

### 4.4 Time-invariant Normalization

Under modest assumptions, we show that the dynamic range of $\bar{\alpha}$ is bounded by linear functions of time. Equation (5) suggests that

$$
\begin{align*}
\max_j \bar{\alpha}_j(t) - \max_i \bar{\alpha}_i(t-1) & \leq \max_i \bar{\alpha}_{ij} + \max_j \bar{b}_j(t) \\
\min_j \bar{\alpha}_j(t) - \min_i \bar{\alpha}_i(t-1) & \geq \min_i \bar{\alpha}_{ij} + \min_j \bar{b}_j(t)
\end{align*}
$$

(13)
Equation (13) can be written recursively for all frames and summed up on both sides, leading to

\[
\begin{align*}
\max_j \bar{a}_j(t) & \leq t \cdot \max_{ij} \bar{a}_{ij} + \sum_{s=1}^t \max_j \bar{b}_j(s) \\
\min_j \bar{a}_j(t) & \geq t \cdot \min_{ij} \bar{a}_{ij} + \sum_{s=1}^t \min_j \bar{b}_j(s)
\end{align*}
\]  

(14)

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

\[
r_l t \leq \bar{a}_j(t) \leq r_h t
\]  

(15)

where

\[
\begin{align*}
r_h & \triangleq \max_{ij} \bar{a}_{ij} + E[\max_j \bar{b}_j(t)]; \\
r_l & \triangleq \min_{ij} \bar{a}_{ij} + E[\min_j \bar{b}_j(t)].
\end{align*}
\]

Motivated by Equation (15), we propose a normalized forward probability \( \eta_j(t) \equiv \phi_j(t)e^{rt} \), where \( r \) is a positive constant. The final Viterbi score consequently becomes

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

(16)

First, Equation (16) is a valid scoring criterion simply 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:

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

(17)

As is shown in Section 7, 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 (15) we have

\[
(r_l + r)t \leq \bar{\eta}_j(t) \leq (r_h + r)t.
\]  

(18)

To choose \( r \), we compute the scores of all utterances from the training set evaluated on their own generative word models, and let \( r = -E[\log P(O_{1:T}|\text{correct model})/T] \), in an attempt to normalize to zero the highest log likelihood score for an utterance (also the correct model’s log likelihood with zero word error). It still might be true that when evaluating utterances with respect to a wrong word model the score decreases as \( T \) increases. When this happens, however, it will be for those words with lower partial likelihoods. The scheme, therefore, is analogous to pruning, where we essentially prune away unpromising partial hypotheses by collapsing their likelihoods down to be encoded with a very few number of bits.

5 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 distance or Mahalanobis distance for vector quantization [19]. However, it is important to investigate new metrics customized to minimize the degradation in recognition accuracy.

As will be shown in Section 8, the forward probability requires the highest bit-width among all system variables and hence has the greatest impact on the system bandwidth and the total table size. We are therefore particularly interested in further compressing this variable. Forward probabilities are in fact just likelihoods. It turns out that different likelihood magnitudes can be either more or less important for generating the ultimate correct answer. 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 regardless of their relative value. A standard data-driven quantization scheme,
However, tends to allocate more bits to a value range only based on its higher probability mass. Since low likelihoods are the more probable (there is only one correct answer), more bits will be allocated to these low scores at quantization time, thereby giving them high resolution. Such a quantization scheme would be quite wasteful.

Therefore, we propose an alternate discriminatively inspired distortion measure to penalize low-valued forward probabilities. We define the distortion between a sample \( \bar{y}_j(t) = x \) and its quantized value \( Q(x) \) as

\[
D(x, Q(x)) = \frac{(x - Q(x))^2}{f(x)},
\]

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.

### 6 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 make the following definitions:

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

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 \( \delta_i \) as the ratio of the decrease in WER to the increase in cost evaluated in the direction of \( bw_i \).

\[
\delta_i(\bar{bw}) \triangleq \frac{\text{wer}(\bar{bw}) - \text{wer}(\bar{bw} + \Delta \bar{bw}_i)}{\text{cost}(\bar{bw} + \Delta \bar{bw}_i) - \text{cost}(\bar{bw})}
\]

Equation (20) reflects the rate of improvement along the \( bw_i \) direction. We can extend this definition to the gradient along multiple directions. For example,

\[
\delta_{ij}(\bar{bw}) \triangleq \frac{\text{wer}(\bar{bw}) - \text{wer}(\bar{bw} + \Delta \bar{bw}_i + \Delta \bar{bw}_j)}{\text{cost}(\bar{bw} + \Delta \bar{bw}_i + \Delta \bar{bw}_j) - \text{cost}(\bar{bw})}
\]

is the gradient in the joint direction of \( bw_i \) and \( bw_j \).

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

\[
\bar{bw}^* = \arg\min_{\bar{bw} : \text{wer}(\bar{bw}) \leq \text{BWER}} \text{cost}(\bar{bw})
\]

This search space is highly discrete and, due to the effects of quantization noise, only approximately smooth. The combination of these factors means the optimization is very difficult. An exhaustive search for \( \bar{bw}^* \), evaluating \( \text{wer}(\bar{bw}) \) and \( \text{cost}(\bar{bw}) \) at every possible \( \bar{bw} \), is clearly exponential in \( L \). Even constraining the bit-width of each variable to

UWEETR-2004-0019 9
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 \( \tilde{bw} \) 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 [21] to optimize floating-point bit-widths.

### 6.1 Single-Variable Quantization

Finding a reasonable starting point is an important part of these algorithms. One logical place to start is with the results of single-variable quantization, since the results of those quantizations are needed in order to create the tables. Specific methods for quantizing individual variables have been introduced in the previous sections.

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 \( M_i \) based on inspection.

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

### 6.2 Single-Dimensional Increment

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

1. Evaluate the gradients \( \delta_i(\tilde{bw}), i = 1..L \), for the current \( \tilde{bw} \) according to Equation (20), where \( L \) tests are needed to obtain the WERs. If \( \delta_i(\tilde{bw}) < 0 \forall i \), return \( \tilde{bw}^* = \tilde{bw} \);

2. Choose the direction \( k = \arg\max_{i:bw_i+1 \leq M_k} \delta_i(\tilde{bw}) \), and set \( \tilde{bw} = \tilde{bw} + \Delta \tilde{bw}_k \);

3. If \( \text{wer}(\tilde{bw}) \leq \text{BWER} \), return \( \tilde{bw}^* = \tilde{bw} \); otherwise repeat steps 1 and 2.

For speech recognition, online evaluation for a test takes a significant amount of time, requiring a full test cycle. 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 might thus become stuck in a local optimum.

### 6.3 Multi-Dimensional Increment

As an improvement, 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 \( L + \left( \frac{L}{2} \right) \) possible candidates. We could extend this to include triplet increments, but it would take an intolerably long time (years) to finish. The algorithm differs from the above one in steps 1 and 2:

1. Evaluate the gradients \( \delta_i(\tilde{bw}) \) and \( \delta_{ij}(\tilde{bw}), i = 1..L, j = 1..L \) on current \( \tilde{bw} \) according to Equations (20) and (21) respectively, where \( L + \left( \frac{L}{2} \right) \) tests are needed to obtain the WERs. Return \( \tilde{bw}^* = \tilde{bw} \) if all these gradients are negative;

2. Choose the direction \( k \) or a pair of directions \( \{k, l\} \) where \( \delta_k(\tilde{bw}) \) or \( \delta_{kl}(\tilde{bw}) \) 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.
3. If $\text{wer}(b\tilde{w}) \leq \text{BWER}$, return $b\tilde{w}^* = b\tilde{w}$; otherwise repeat steps 1 and 2.

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. Due to the measures it takes to avoid local optima, this algorithm takes substantially longer to complete. Again, this is acceptable since it happens during the design stage, not during recognition.

7 System Organization

7.1 Baseline System Configuration

The database used for system evaluation is NYNEX PhoneBook [22], 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 $\mu$-Law. We set aside 79778 utterances for training, 6598 for development and 7191 for evaluation. 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 noisy 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 concatenated together into words based on pronunciation. The pronunciation models determine the transition probabilities between phones. Doing this provides a customizable vocabulary without the need for retraining, a desirable goal for an embedded device. Our system has 42 phone models, each composed of 4 emitting states except for the silence model. 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 $\mu$-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 [23], where we apply pre-emphasis to each frame, multiply it with a hamming window, take the FFT, then calculate the log energy of the mel frequency bins, and finally perform a DCT to obtain the 13-dimensional static-feature vectors. We then add the first order dynamic features. 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.

7.2 Codebook and Table Definition

Based on the analysis in the previous sections, we defined 13 variables to be quantized which are listed in Table 1. The variable $x$ is the output feature element of the front-end, $m$, $v$, $c$, $w$ and $a$ are the acoustic model parameters pre-computed, and $s$, $d$, $e$, $p$, $q$, $b$ and $\eta$ are other intermediate variables in the back-end system.

2Specifically, the training set consisted of subsets aa, ab, ah, ai, an, aq, at, au, ax, ba, bb, bh, bi, bm, bn, bq, bt, bu, bx, ca, cb, 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, ce, do, dy.
Table 1: System variables and their corresponding expressions in Equation (3)

<table>
<thead>
<tr>
<th>symbol</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$x_k(t)$</td>
</tr>
<tr>
<td>m</td>
<td>$\mu_{i,k}$</td>
</tr>
<tr>
<td>v</td>
<td>$\sigma_{i,k}^2$</td>
</tr>
<tr>
<td>c</td>
<td>$c_i$</td>
</tr>
<tr>
<td>w</td>
<td>$\tilde{w}_i$</td>
</tr>
</tbody>
</table>

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

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

$F_1(\cdot)$ 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 (17) are easily implemented, since they can be achieved using low bit-width integer comparisons operations, so no extra tables are required.

8 Experiments and Results

This section first reports the results of system development. The LBG [24] 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.

8.1 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 2 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 $e$ and the forward probability $\eta$ which will be discussed separately later. Note that we let $q$ and $b$ share the same codebook because their value ranges have much overlap.

Table 2: Minimum bit-width to which each variable can be individually quantized without increase in WER

<table>
<thead>
<tr>
<th>variable</th>
<th>m</th>
<th>v</th>
<th>w</th>
<th>a</th>
<th>x</th>
<th>s</th>
<th>d</th>
<th>c</th>
<th>p</th>
<th>b (q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>min bit</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Section 3 presented two approaches to quantize variable $e$. In our case, the operation $e = e + d$ is repeated 26 times to get the final value of $e$. Using linear accumulation, it involves only two variables $e$ and $d$ and only one table
$e = e + d$. Alternatively, we can use tree-structure accumulation with multiple variables and $\lfloor \log 26 \rfloor = 5$ 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 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 $e$ is involved. Assuming $n_d = 5$, $n_c = 5$ and $n_p = 6$ according to Table 2, 6 kBytes of LUTs are needed to realize the operations involving $e$ 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 $e$, 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.

![Figure 3: Single-variable quantization for accumulative variable $e$](image)

Section 4 proposed a time-invariant normalization to the forward probability to reduce its entropy. 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, the discriminative distortion measure works slightly better than the normal one.

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 3. Recall that the baseline WER of the development set is 2.07%.

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.

---

3If $e$, $d$, $c$ and $p$ are quantized to $n_e$, $n_d$, $n_c$ and $n_p$ bits respectively, the total size of related tables is $n_e 2^{n_d+n_e} + n_p 2^{n_e+n_p}$ bits.
Figure 4: Single-variable quantization for forward probability

Table 3: Comparison of two bit-width optimization algorithms

<table>
<thead>
<tr>
<th></th>
<th>WER</th>
<th>Table (kB)</th>
<th># Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-D</td>
<td>3.09%</td>
<td>36.44</td>
<td>52</td>
</tr>
<tr>
<td>2-D</td>
<td>2.53%</td>
<td>58.75</td>
<td>454</td>
</tr>
</tbody>
</table>

The final bit-width allocation scheme of the last algorithm is shown in Table 4, where the first row indicates the variable and the second row shows the corresponding bit-width. As 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.

Table 4: The optimal bit-width allocation scheme

<table>
<thead>
<tr>
<th>V</th>
<th>m</th>
<th>v</th>
<th>w</th>
<th>x</th>
<th>s</th>
<th>d</th>
<th>e</th>
<th>c</th>
<th>p</th>
<th>b</th>
<th>a</th>
<th>( \eta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>BW</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>5</td>
<td>3</td>
<td>9</td>
</tr>
</tbody>
</table>

8.2 Final system evaluation

We tested the recognition performance of the scheme in Table 4 on our evaluation set for all 75-, 150-, 300- and 600-word vocabulary cases as defined in Section 7. 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 5, 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 — the decision boundaries are probably slightly less susceptible to minor perturbations when variables are so coarsely quantized in the custom arithmetic case.

<table>
<thead>
<tr>
<th>vocab.</th>
<th>75-word</th>
<th>150-word</th>
<th>300-word</th>
<th>600-word</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fp.</td>
<td>custom</td>
<td>fp.</td>
<td>custom</td>
</tr>
<tr>
<td>clean</td>
<td>2.26</td>
<td>2.99</td>
<td>3.23</td>
<td>4.43</td>
</tr>
<tr>
<td>30 dB</td>
<td>2.68</td>
<td>3.46</td>
<td>3.91</td>
<td>4.98</td>
</tr>
<tr>
<td>20 dB</td>
<td>8.55</td>
<td>8.23</td>
<td>11.71</td>
<td>11.13</td>
</tr>
<tr>
<td>10 dB</td>
<td>30.65</td>
<td>29.59</td>
<td>38.36</td>
<td>36.87</td>
</tr>
</tbody>
</table>

8.3 **CPU Time Simulation**

We utilized SimpleScalar [25], 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, a detailed 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 to reflect more current values. The goal of this is not to make performance predictions based on specific numbers but to see whether the approach scales well. 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. The 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.

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 4, 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.

9 **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 entropies, 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 NSF for funding this project under Award-0086032 and thank Carl Ebeling for useful advice.

References


