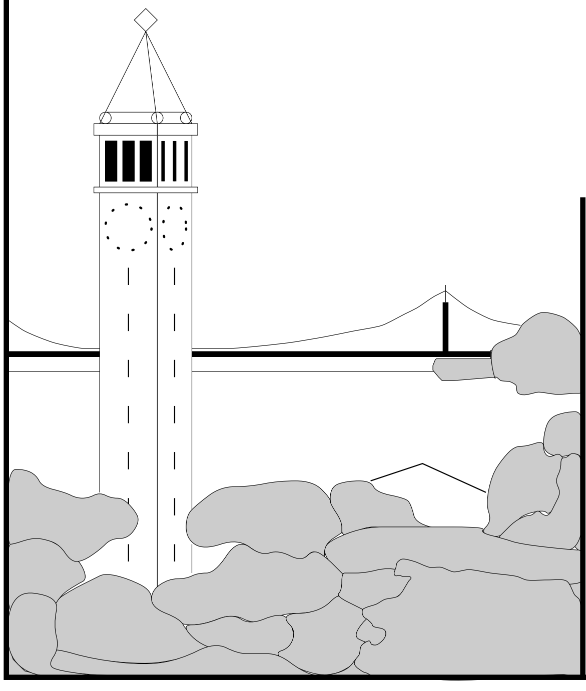
Dynamic Memory Model based Optimization of Scalar and Vector Quantizer Encoder

Gene Cheung and Steven McCanne



Report No. UCB/CSD-99-1085

February 28, 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Work supported by Grant
FD97-34515-McCanne(ANI-9734515)

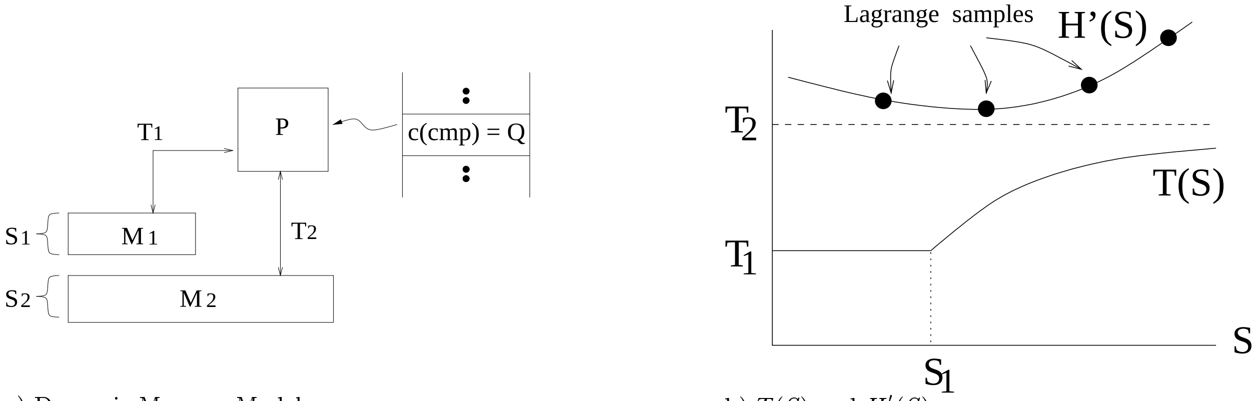
Abstract

The rapid progress of computers and today's heterogeneous computing environment means computation-intensive signal processing algorithms must be optimized for performance in a machine dependent fashion. In this paper, we present design and analysis of an automated algorithm optimizer for scalar and vector quantizer encoders. Using a dynamic memory model, the optimal computation-memory tradeoff is exploited to minimize the encoding time. Experiments show our proposed optimized algorithm has marked improvements over existing techniques.

1 Introduction

If the computer evolution has matured to a stage where computers are ubiquitous and homogeneous, and improvements are asymptotic, then implementation of a signal processing algorithm needs only be painstakingly hand-coded once for optimal performance. Unfortunately, computers continue to progress at an exponential rate, and computing environments are extremely diverse. Clearly, hand-coding an algorithm for every possible platform is impractical. On the other hand, simply compiling a fixed algorithm written in a high level language for each machine is sub-optimal, since machine dependent information such as memory hierarchy is unexploited at the algorithmic level. A fundamental question surfaces: how to re-target an algorithm onto different machine platforms optimally *and* automatically?

In light of this problem, recent research [1], [2] has looked at the distortion-computation tradeoffs of particular algorithms, thus providing formal analyses of algorithm tuning for machines with different computational budgets. Our work [3, 4, 5] differs from previous work in that instead of minimizing the number of computational units for a given distortion, we search for the optimal computation-memory tradeoff to minimize running time: divide the processing so that the optimal subset is implemented as simple data memory retrievals of pre-computed values (*pre-compute*), and the other is implemented as on-the-fly computations (*compute*). *pre-compute* requires only a single data memory lookup but may lead to memory blow-up; *compute* can be slow but avoids the memory implosion problem. The tension between compute and pre-compute is an interesting one, and if it



a) Dynamic Memory Model b) $T(S)$ and $H'(S)$

Figure 1: Machine Model and Optimization Framework

is correctly exploited for a given machine, can lead to enhanced performance over techniques that completely ignore one or the other. In this paper, we will demonstrate this is indeed the case for scalar and vector quantizer encoding algorithms.

The importance of quantization is paramount, as just about every compression algorithm includes quantization of some form. But while scalar quantizer is widely used, vector quantizer is less accepted, partly due to its inherent high encoding complexity. In this paper, we show that by exploiting the computation-memory tradeoff for a particular machine, we can improve encoding performance of both quantizers. In section 2, we first review the machine model and its associated optimization framework in [4]. In section 3 and 4, we discuss how the framework is instantiated for scalar and vector quantizer encoders respectively. We present results in section 5. Finally, we conclude in section 6.

2 Dynamic Memory Model and Optimization Framework

2.1 Dynamic Memory Model

Modern processors use hierarchical memories to enhance performance, where small, fast memories are located near the CPU and larger, slower memories are situated further away. Consequently, the execution speed of a machine instruction that accesses memory depends on the level of memory referenced. The machine model in Figure 1a reflects this characteristic. If the processor P accesses a datum residing in level 1 memory $\mathbf{M}_1$ (level 2 memory $\mathbf{M}_2$), it incurs memory access time T_1 (T_2). If the instruction does not involve memory access, then the execution time depends on the complexity of the instruction itself; we denote the cost of a logical comparison (*cmp*) as Q. We assume for now the sizes of $\mathbf{M}_1$ ($\mathbf{M}_2$) is S_1 (∞)[1].

Suppose the size of data structures of an algorithm in memory, S, is $\leq S_1$. Then the access time of a desired datum, $T(S)$, is T_1, since all data structures can be loaded into $\mathbf{M}_1$. If $S > S_1$, then the exact location of the desired datum is hard to tract; depending on the processor's caching policy and data access patterns, it can be in $\mathbf{M}_1$ or $\mathbf{M}_2$. In this case, we estimate the access time as (see Figure 1b):

$$T(S) = \begin{cases} T_1 & \text{if } S \leq S_1 \\ (\frac{S_1}{S})T_1 + (\frac{S-S_1}{S})T_2 & \text{otherwise} \end{cases} \tag{1}$$

The basic idea is the following: assuming all pieces of data are equally probable, with probability $\frac{S_1}{S}$ ($\frac{S-S_1}{S}$) we will find a datum in $\mathbf{M}_1$ ($\mathbf{M}_2$) with access time T_1 (T_2).

2.2 Optimization Framework

Using the dynamic memory model, we evaluate the execution cost of an algorithm l as follow. We first find the size of the algorithm's data structures, $R(l)$. This translates to a memory access cost $T(R(l))$ using (1). Knowing the access cost, we can evaluate the execution cost of l, $H_{T(R(l))}(l)$. Let $\mathcal{L}$ denote the set of algorithms in the search space. The optimization problem is:

$$\min_{l \in \mathcal{L}} \left\{ H_{T(R(l))}(l) \right\} \tag{2}$$

Solving (2) is difficult in general (see [4] for VLC decoding example). The reason is twofold: i) while the cost of a memory access is not known till the entire algorithm is constructed, the optimal

[1] We can easily generalized the memory model to any number of memory levels of any size.

construction of an algorithm depends on the cost of memory access — a classic chicken-and-egg problem; ii) dependency on non-linear function T(S) means the optimization problem is non-linear. Instead of solving (2) directly, we first dissect it into easier pieces.

Suppose we know *a priori* that the total data structure size of the optimal algorithm l^* in memory is S^*. To find l^*, we only need to search the subset of algorithms with total size S^*. Let $H_{T(S)}(l)$ denote the cost of l when the access cost is fixed at $T(S)$. (2) is then the same as:

$$H'(S^*) = \min_{l \in \mathcal{L}} \{H_{T(S^*)}(l)\} \quad \text{ s.t. } R(l) = S^* \tag{3}$$

Solving $H'(S^*)$ seems easier, since the mutual dependency and the non-linearity have both been removed. The problem is we do not know S^* a priori, and so we need to search through all S values for S^*:

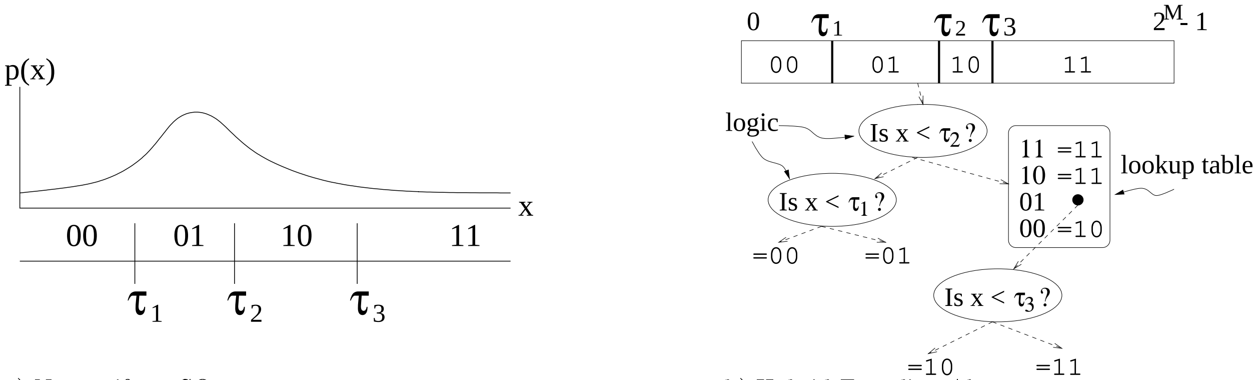
$$\min_{l \in \mathcal{L}} \{H_{T(R(l))}(l)\} = \min_{\forall S} \{H'(S)\} \tag{4}$$

See Figure 1b for an illustration. We are now faced with two new difficulties: i) solving (3) for all S is expensive, ii) (3) itself is still hard since it is a constrained problem. The idea is not to solve (3), but solve its corresponding Lagrangian:

$$\min_{l \in \mathcal{L}} \{H_{T(S)}(l) + \lambda R(l)\} \tag{5}$$

It is proven in [4] that by finding a suitable multiplier λ, if the optimal solution to (5), l^o, is such that $R(l^o) = S$, then l^o is also optimal to (3). We accomplish that by applying the following iteration: we adjust λ so that $R(l^o)$ is as close to S as possible while keeping $R(l^o) \geq S$. If $R(l^o) = S$, then $H_{T(S)}(l^o)$ is a valid point on $H'(S)$. If not, we let $S = R(l^o)$, repeat the above procedure until $R(l^o) = S$.

Notice that after applying the iteration, only a subset of possible S values has the corresponding Lagrangian solutions satisfying $R(l^o) = S$. But in so doing, we are lowering the complexity by only sampling a small number of points on $H'(S)$. We call this phenomenon *Lagrangian sampling* (see Figure 1b). By sampling, however, we may not be able to find the optimal $H'(S^*)$, and we rely on a sampling error theorem in [4] to bound our solution error.



a) Non-uniform SQ

b) Hybrid Encoding Algo.

Figure 2: Non-uniform SQ and Encoding Algorithm

3 Scalar Quantizer

The particular scalar quantizer (SQ) we are concerned with is a non-uniform SQ for a sequence of already digitized M-bit fixed point inputs. The optimal design of non-uniform SQs is well-studied [7]; the resulting N-bit quantizer is commonly called the *Lloyd-Max Quantizer*. In short, the quantizer finds the optimal 2^N partitions for all input values — partition boundaries denoted by $\Gamma = \{\tau_1, \ldots, \tau_{2^N-1}\}$ — given input probability $p(x)$. An example is shown in Figure 2a. The problem is to implement the M-to-N-bit non-uniform quantizer as efficiently as possible.

Lets first consider two simple encoding algorithms. The first one minimizes encoding steps by performing a single M-bit table lookup, where the resulting entry contains the corresponding N-bit partition index. However, this requires 2^M memory storage, and memory access can be slow if M is large and 2^M elements cannot fit into $\mathbf{M}_1$. An alternative algorithm minimizes memory usage by asking a sequence of logic statements "*Is* $x < \tau_i$*?*" until the correct partition has been identified. This corresponds to a binary decision tree of height $h \geq N$. If N is large, the sequence of questions required is long and the algorithm is slow. A natural question is: what is the optimal hybrid scheme, using a combination of lookup tables and logic, that minimizes the encoding time? An example is shown in Figure 2b.

To correctly exploit the tradeoff between memory and computation, we will use the dynamic memory model and framework. Essentially, we want to instantiate (5) for the SQ problem, given threshold set $\Gamma = \{\tau_1, \ldots, \tau_{2^N-1}\}$ and input probability $p(x)$.

3.1 Algorithm Development

Let $f(a,b)$ be the minimum Lagrangian encoding cost — the value of (5) — given input $x \in [a,b)$. The optimal operation for this input range can potentially be a logic or table lookup, resulting in cost $f_l(a,b)$ or $f_t(a,b)$ respectively:

$$f(a,b) = \min\{\ f_l(a,b),\ f_t(a,b)\ \} \tag{6}$$

For logic, we can choose among all τ_i values that is in range (a,b) to check against input x. The result of the check is a partition of original interval into $[a,\tau_i)$ and $[\tau_i,b)$. Let $p(a,b)$ denote the probability that $x \in [a,b)$, and Q denote the cost of a logic operation. We can write $f_l(a,b)$ as:

$$f_l(a,b) = p(a,b)Q + \min_{\tau_i \in (a,b)} [f(a,\tau_i) + f(\tau_i,b)] \tag{7}$$

For table lookup, there is first an access cost of $p(a,b)T(S)$. The table lookup operation is an index operation into a table using left-most h-bit of index $x-a$. The number of bits needed to define range $b-a$, thus the maximum height of a lookup table, is $\lceil \log_2(b-a) \rceil$. For each table height h, the table operation divides the range $[a,b)$ into smaller ranges of width $m = 2^{\lceil \log_2(b-a) \rceil - h}$ each. The number of these smaller ranges, n, is determined by the largest number the h most significant bits of $b-a$ can take on. The associated penalty $\lambda R(l)$ in (5) is therefore λn. The following equations formalize this analysis:

$$\begin{aligned} f_t(a,b) &= p(a,b)T(S) + \min_{1 \le h \le \lceil \log_2(b-a) \rceil} \left[\lambda n + \sum_{i=1}^{n-1} f(a+m(i-1), a+m(i)) + f(a+m(n-1), b) \right] \\ m &= 2^{\lceil \log_2(b-a) \rceil - h} \qquad\qquad n = \left\lfloor \frac{b-a-1}{m} \right\rfloor + 1 \end{aligned} \tag{8}$$

The base case of the recursion is when there is no τ_i in range (a,b), meaning the input x can only be in one partition:

$$f(a,b) = 0 \qquad \text{if} \quad \nexists\ \tau_i \in (a,b) \tag{9}$$

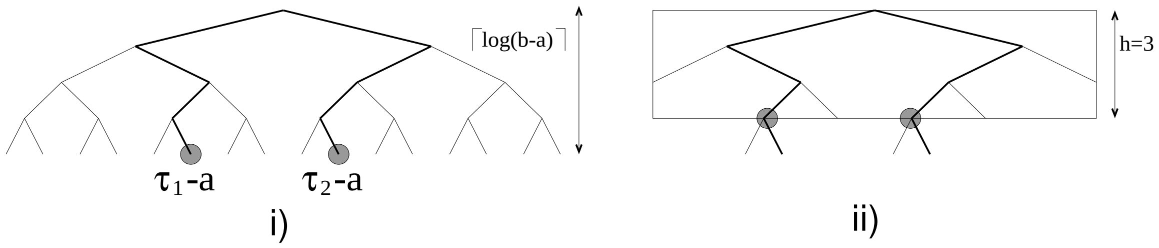


Figure 3: Tree Pruning Example

Recursive call to $f(0, 2^M)$ yields the optimal solution to (5), l^o, given λ.

3.2 Tree Pruning

Looking at (8), we notice that the algorithm have 2^M recursive calls when height M lookup table is tested for $f_t(0, 2^M)$. This means a call to (8) has running time $O(2^M M)$ — running time is exponential. (Recall that the inputs of the algorithm are Γ and $p(x)$, where $|\Gamma| = 2^N$, $N << M$.) However, we can reduce the complexity with the following observation. When performing an h-bit table lookup operation for a given range $[a, b)$ using index $x - a$, among the n branches the input may fall into, we only need to further check which partition x falls into for the branches that correspond to $\tau_i - a \in (0, b - a)$. See Figure 3 for an example. Notice that unless x falls into one of these *τ-branches*, we know instantly without any more operations which partition input x falls into. Therefore, the only recursive calls needed in (8) are these τ-branches. The following equation expresses this idea:

$$\begin{aligned} f_t(a,b) &= p(a,b)T(S) + \\ &\quad \min_{1 \le h \le \lceil \log_2(b-a) \rceil} \left[\lambda n + \sum_{\tau_i \in (a,b)} \mathbf{1}(i \neq n) f(a + m(i-1), a + m(i)) + \mathbf{1}(i = n) f(a + m(n-1), b) \right] \\ m &= 2^{\lceil \log_2(b-a) \rceil - h} \qquad\qquad n = \left\lfloor \frac{b-a-1}{m} \right\rfloor + 1 \\ i &= \left\lfloor \frac{\tau_i - a}{m} \right\rfloor + 1 \end{aligned} \tag{10}$$

where $\mathbf{1}(p)$ is the indication function — evaluates to 1 if predicate p is true, 0 otherwise. The complexity of computing (10) is now $O(2^N M)$.

3.3 Dynamic Programming

It can be easily shown that when solving $f(0, 2^M)$ using (7) and (10), there are overlapping sub-problems. We can eliminate the overhead of solving the same sub-problem more than once by storing each sub-problem's answer in a *dynamic programming table*. Each time we encounter a sub-problem $f(a, b)$, we first check the table entry corresponding to argument (a, b) to see if it has been solved before. If so, we simply return the value. If not, we solve it using (7) and (10) and store the value in that entry location for possible future lookups. Since the completion of the dynamic programming table means the largest sub-problem — the original problem $f(0, 2^M)$ — has been solved, the complexity of the algorithm will be the size of the dynamic programming table times the complexity of computing each table entry. The complexity of computing each entry using (7) and (10) is: $O(2^N) + O(2^N M) = O(2^N M)$. We now investigate the minimally sufficient size of the dynamic programming table.

We first note that creating a table of size $2^M * 2^M$ is clearly impractical — it would mean algorithm requires exponential memory size. To restrict table size, we first look closely at how arguments (a, b) take on different values as (7) and (10) are called recursively. Looking at the two equations, we can summarize the evolution of these values using the following context-free grammar rules:

$$\begin{array}{lllllll} a & \rightarrow & a + (i-1) * 2^h & \qquad & b & \rightarrow & a + j * 2^h \\ a & \rightarrow & \tau_i & & b & \rightarrow & \tau_j \end{array} \tag{11}$$

From the above rules, it is clear that a can be written as:

$$a = \tau_i + (i_1 - 1) * 2^{h_1} + (i_2 - 1) * 2^{h_2} + \ldots + (i_p - 1) * 2^{h_p} \tag{12}$$

$$= \tau_i + (i-1) * 2^{h_i} \tag{13}$$

where $h_i = \min[h_1, h_2, \ldots, h_p]$. Note that i must take on values so that there is a τ value in range $(\tau_i + (i-1) * 2^{h_i},\ \tau_i + i * 2^{h_i})$; the number of these i values is $O(2^N)$. For each possible τ_i in (13), h_i

can take on $O(M)$ possible values, and there are at most 2^N possible τ_i values. Hence the possible number of values for a is $O(2^{2N}M)$. A similar analysis will show the possible values for b is also $O(2^{2N}M)$. Assuming we can retrieve an entry corresponding to $f(a,b)$ in constant time, we can conclude the complexity of the algorithm is: $O(2^{2N}M * 2^{2N}M * 2^N M) = O(2^{5N}M^3)$.

3.4 Singular Value Search

3.4.1 General Theory of Singular Values

In the previous two sections, we discussed in details how to solve the SQ instantiation of (5) of the optimization framework. The algorithm is described by (7) and (10) — the algorithm solves (5) for a given multiplier value λ. Recall that if the optimal Lagrangian solution l^o is such that $R(l^o) \neq S$, we need to adjust the multiplier value and solves (5) again and again, until $R(l^o)$ is as close to S as possible while keeping $R(l^o) \geq S$.

Consider the example in Figure 4. The Lagrangian cost of every algorithm in search space $\mathcal{L}$ is represented as a linear function of multiplier λ, $H(l) + \lambda R(l)$, in the top graph. The bottom graph plots the slope of the optimal Lagrangian cost function given λ, or simply $R(l^o)$. Notice as λ increases, the optimal algorithm changes from l_1 to l_2 to l_3, and the corresponding $R(l^o)$ changes from θ_1 to θ_2 to 0. Note also that algorithm l_4 is never the optimal algorithm for any multiplier value.

Notice at special multiplier value, λ_1, λ_2 in Figure 4, there are two algorithms that are simultaneously optimal; for example, at λ_1, algorithm l_1 and l_2 are both optimal. These multiplier values are termed *singular values* in [6]. It turns out by solving (5) *only* at singular multiplier values, we can discover all solutions to (5) for all multiplier values. Moreover, at the particular multiplier value where the slopes of the two optimal solutions span the constraint S, we can conclude that these two are the solutions with $R(l)$ closest to S from above and below, among all Lagrangian solutions. In

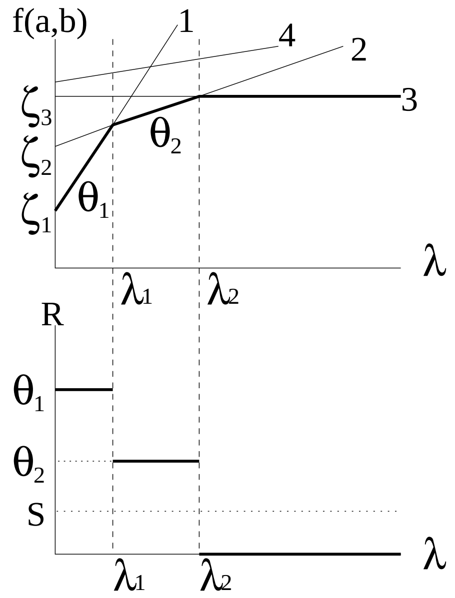


Figure 4: Cost as Function of Multiplier Value

the example, at λ_2, l_2 and l_3 are simultaneously optimal, and given they are the closest Lagrangian solutions from above and below, l_2 is the solution we terminate with in the singular value search iteration.

3.4.2 SQ Instantiation

We now turn our attention to the problem of finding the neighboring singular value for the SQ problem. We assume we have already obtained the optimal Lagrangian solution to (5) given $\lambda = \lambda^o$ by solving $f(a,b)$. Let $\zeta(a,b)$ ($\theta(a,b)$) be the y-intercept (slope) of the optimal Lagrangian solution given input $x \in [a,b)$, i.e. $f(a,b) = \zeta(a,b) + \lambda^o \theta(a,b)$. These values can be stored in dynamic programming tables as $f(a,b)$ is being solved, thus incurring no addition computational cost. We now define a related function $g(a,b)$ that returns the larger neighboring singular multiplier value given input $x \in [a,b)$ — the minimum value at which the current optimal algorithm will become co-optimal with a new algorithm as λ increases. We call it the *augmented multiplier*, denoted by λ^+. The new algorithm may use a logic or a table for range $[a,b)$, so we will check both cases and find the minimum of the two. Let $g_l(a,b)$ ($g_t(a,b)$) be a function that returns λ^+ given input $x \in [a,b)$ and given the new algorithm's optimal operator for range $[a,b)$ is logic (table). We can write $g(a,b)$ as:

$$g(a,b) = \min\{\ g_l(a,b),\ g_t(a,b)\ \} \tag{14}$$

For $g_l(a,b)$, we assume the new optimal algorithm l' has logic as the optimal operator for range $[a,b)$. Suppose we further assume the optimal solutions of the corresponding children for $\lambda = \lambda^o$ remains the same as λ increases. Then Langragian cost of l' is:

$$H(l') + \lambda R(l') = [p(a,b)Q + \zeta(a,\tau_i) + \zeta(\tau_i,b)] + \lambda\,[\theta(a,\tau_i) + \theta(\tau_i,b)] \tag{15}$$

If l' is indeed the next optimal algorithm at augmented value λ^+, its cost function will have to intersect l at λ-value $> \lambda^o$. We will find this intersection point by a call to $X_{(a,b)}[\zeta,\theta]$, which returns the intersection λ-value between optimal Lagrangian solution of $f(a,b)$ and line with slope θ and intercept ζ. If the intersection λ-value is $\leq \lambda^o$, then it returns ∞.

Previously we assume the optimal solutions for $\lambda = \lambda^o$ of the corresponding children remains the same as λ increases. We will need to check this is indeed the case; we check by recursively calling $g(a,\tau_i)$ and $g(\tau_i,b)$ and finding the minimum of these two. If the returned minimum is smaller than the previously calculated intersection point, then we will use this minimum as $g_l(a,b)$ instead. $g_l(a,b)$ can now be written as follow:

$$g_l(a,b) = \min \begin{cases} \min\limits_{\tau_i \in (a,b)} \{\ \min\ [\ g(a,\tau_i),\ g(\tau_i,b)\ ]\ \} \\ \min\limits_{\tau_i \in (a,b)} \ X_{(a,b)} \begin{bmatrix} p(a,b)Q + \zeta(a,\tau_i) + \zeta(\tau_i,b) \\ \theta(a,\tau_i) + \theta(\tau_i,b) \end{bmatrix} \end{cases} \tag{16}$$

Note the similarity between (16) and (7). In fact, following the same complexity analysis, they have the same order of complexity.

We now derive $g_t(a,b)$, and as one would expect, it is similar to $f_t(a,b)$. Here we assume the new algorithm's optimal operation for range $[a,b)$ is a table of some height h. Again, we first assume the optimal solutions at λ^o of the corresponding children remains optimal as λ increases. For table of height h, the new algorithm will now have y-intercept and slope as follow:

$$\text{y-intercept} = p(a,b)T(S) + \sum_{\tau_i \in (a,b)} \begin{bmatrix} \mathbf{1}(i \neq n)\zeta(a+(i-1)m, a+im)+ \\ \mathbf{1}(i = n)\zeta(a+(n-1)m, b) \end{bmatrix} \tag{17}$$

$$\text{slope} = n + \sum_{\tau_i \in (a,b)} \begin{bmatrix} \mathbf{1}(i \neq n)\theta(a+(i-1)m, a+im)+ \\ \mathbf{1}(i = n)\theta(a+(n-1)m, b) \end{bmatrix} \tag{18}$$

where i, n, m are similarly defined in (10). Similar to $g_l(a,b)$, we will need to check our assumption of continued optimality of corresponding children are valid. We will again do so using recursive calls. $g_t(a,b)$ can now be written as:

$$g_t(a,b) = \min \begin{cases} \min\limits_{\substack{1 \leq h \leq \lceil \log_2(b-a) \rceil \\ \tau_i \in (a,b)}} \begin{cases} \mathbf{2}(i \neq n) g(a+(i-1)m, a+im) \\ \mathbf{2}(i = n) g(a+(n-1)m, b) \end{cases} \\ \min\limits_{\substack{1 \leq h \leq \lceil \log_2(b-a) \rceil \\ \tau_i \in (a,b)}} X_{(a,b)} \begin{bmatrix} p(a,b)T(S) + \sum\limits_{\tau_i \in (a,b)} \begin{matrix} \mathbf{1}(i \neq n)\zeta(a+(i-1)m, a+im)+ \\ \mathbf{1}(i = n)\zeta(a+(n-1)p, b) \end{matrix} \\ n + \sum\limits_{\tau_i \in (a,b)} \begin{matrix} \mathbf{1}(i \neq n)\theta(a+(i-1)m, a+im)+ \\ \mathbf{1}(i = n)\theta(a+(n-1)m, b) \end{matrix} \end{bmatrix} \end{cases} \tag{19}$$

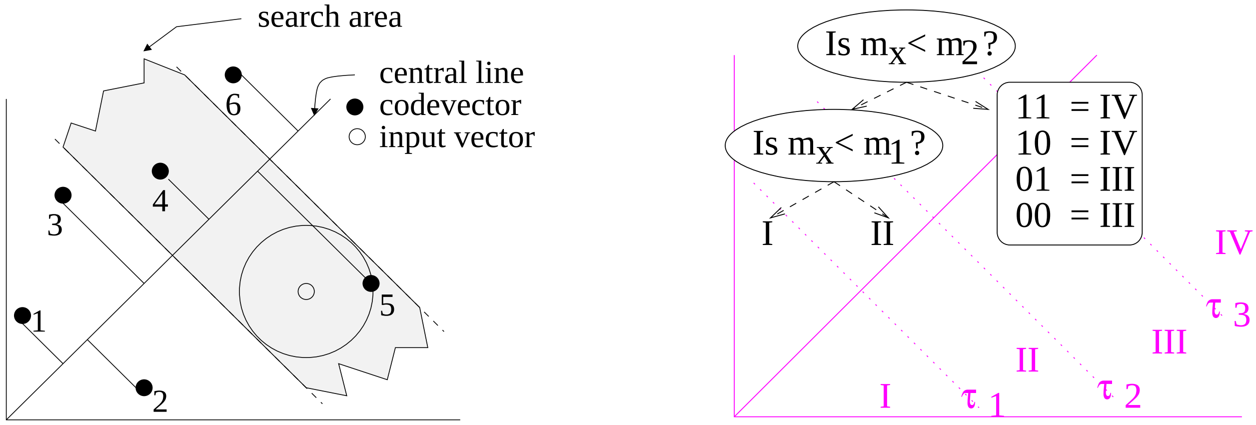
where $\mathbf{2}(c)$ means the function is called only if clause c is true.

Using the same complexity analysis as the one for $f(a,b)$, it is easy to see that the $g(a,b)$ has the same time complexity $O(2^{5N}M^3)$. To find optimal solution l^o to (5) with $R(l^o)$ closest to S

4 Vector Quantizer

VQ encoding is the process of finding the codevector $\mathbf{y}_j$ among codevector set $\mathcal{Y} = \{\mathbf{y}_1, \ldots, \mathbf{y}_N\}$, that minimizes distortion metric $d(\mathbf{x}, \mathbf{y}_j)$ given input vector $\mathbf{x}$:

$$\mathbf{y}_j = \arg\min_{\mathbf{y}_i \in \mathcal{Y}} d(\mathbf{x}, \mathbf{y}_i) \tag{20}$$



a) Equal-avg VQ Encoding

b) Hybrid Encoding Algo.

Figure 5: Equal-average VQ and Hybrid Algorithm

where $\mathbf{x}$, $\mathbf{y}_j$'s are vectors in k-dimension space. This *nearest neighbor search* requires searching through all codevectors in the worst case, if the codebook $\mathcal{Y}$ is unstructured.

4.1 Heuristic Approach

Recently, heuristics such as equal-average hyperplane partition (EAHP) [8] have been shown to lower complexity in the average case for image data. The key observation of EAHP is that there is strong correlations among input vector's individual components for image data. As a result, the majority of the input vectors are distributed along the central line $l = \{\mathbf{x} | x_1 = \ldots = x_k\}$. If we sort the codevectors according to their means, we can successively eliminate potential nearest neighbors by using this bound[2]:

$$d(\mathbf{x}, \mathbf{y}) \geq \sqrt{k} |m_{\mathbf{x}} - m_{\mathbf{y}}| \tag{21}$$

For example in Figure 5a, we first test codevector $\mathbf{y}_5$ and compute $d(\mathbf{x}, \mathbf{y}_5)$. We can then eliminate any vector $\mathbf{y}_i$ whose mean $m_{\mathbf{y}_i}$ is such that $\sqrt{k}|m_{\mathbf{x}} - m_{\mathbf{y}_i}| \geq d(\mathbf{x}, \mathbf{y}_5)$. Geometrically, we eliminate all codevectors that lie outside the gray strip that encloses the circle in Figure 5a. In this example, we eliminate $\mathbf{y}_1$, $\mathbf{y}_2$, $\mathbf{y}_3$ and $\mathbf{y}_6$.

It is important to start EAHP with a codevector that has mean close to the input vector. To

[2]Bound is valid only if the distortion metric is l_2 norm.

this end, EAHP uses a binary decision tree to first find the codevector with the closest mean to the input vector. To speed up this initial search, we use the algorithm discussed in section 3 to classify the input mean $m_{\mathbf{x}}$ into the correct partition. We will show in the results section that the speedup of the initial search for closest vector mean does have noticeable improvement to the overall encoding process.

5 Implementation and Results

5.1 Implementation Issues

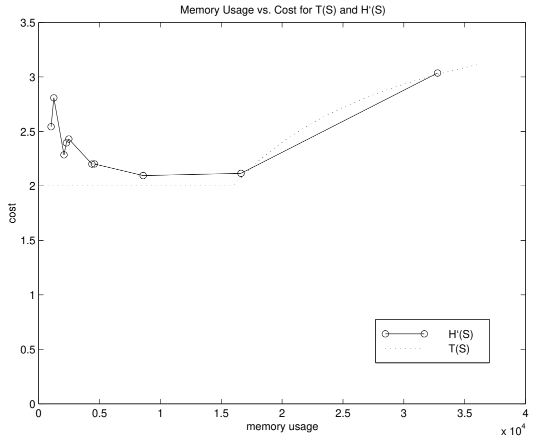
We conjecture that in practice there are more overlapping subproblems when solving $f(a,b)$ that escape our analysis in section 3.3. So in our implementation, instead of constructing a size $2^{2N}M$ * $2^{2N}M$ dynamic programming table for $f(a,b)$, $g(a,b)$, $\zeta(a,b)$ and $\theta(a,b)$, we implemented a 2-dimensional red-black tree[9] of objects, each of which stores all four previously mentioned values corresponding to argument (a,b). To see if $f(a,b)$ has been previously solved, we first use a to search the red-black tree in one dimension. If a is found at a node in dimension one, we use b to search the subtree rooted at this node in dimension two — all of the objects in this subtree has a as first argument. Red-black properties can be preserved in both dimensions if insertions and tree-rotations are done carefully. The resulting implementation means the retrieval of an object corresponding to argument (a,b) is $O(\log a) + O(\log b) = O(M)$.

5.2 Results

To test our algorithm for SQ, we use parameters in Figure 6a and generated $T(S)$ (bottom) and $H'(S)$ for 15-to-4 bit scalar quantizer, with input distribution Gaussian 2 (top) in Figure 6b. The threshold set Γ is generated using Lloyd's algorithm (See Appendix A). The implementation platform is a pentium II 266 MHz processor, with L1 cache 16kbyte (50-50 split of the 32kbyte

Parameters	Values
S_1	$16k$
T_1	2 cycles
S_2	∞
T_2	4 cycles
Q	3 cycles
$p(x)$	Gaussian 1: $N(8000, 400^2)$
	Gaussian 2: $N(8000, 1600^2)$

a) Parameters



b) exp. $T(S)$ and $H'(S)$

Figure 6: Experiment for SQ

p(x)	M	N	algorithm	speed
Gaussian 1	15	2	logic-only	3.947 mil/s
Gaussian 1	15	2	hybrid	4.651 mil/s
Gaussian 1	15	4	logic-only	2.469 mil/s
Gaussian 1	15	4	hybrid	4.545 mil/s
Gaussian 2	15	2	logic-only	3.738 mil/s
Gaussian 2	15	2	hybrid	4.790 mil/s
Gaussian 2	15	4	logic-only	2.484 mil/s
Gaussian 2	15	4	hybrid	4.597 mil/s

a) SQ Encoder Comparison

codebook size	algorithm	encoding time
8	logic-only	.280s/lena
8	hybrid	.255s/lena
16	logic-only	.465s/lena
16	hybrid	.440s/lena
32	logic-only	.465s/lena
32	hybrid	.440s/lena
64	logic-only	.935s/lena
64	hybrid	.895s/lena

b) VQ Encoder Comparison

Figure 7: Comparison of SQ, VQ Encoders

data-instruction cache). To compare our hybrid table lookup-logic SQ encoder to a binary decision tree SQ encoder, we generated 20 million inputs according to the input distribution and encode them 10 times to find an average speed for each case. For input distribution Gaussian 1 15-to-2 bit (15-to-4 bit) SQ encoders, excluding I/O access time, we see a 17.84% speed improvement (84.08%) over logic-only encoder. For input distribution Gaussian 2 15-to-2 bit (15-to-4 bit) SQ encoders, we see a 28.14% improvement (85.06%) over logic-only encoder. Notice that as N increases, the improvement of hybrid encoders over logic-only encoders increases. This is expected, since the height of the binary decision tree for logic-only encoders is larger when as N increases.

For VQ, using 512*512 gray scale images of `Lena`, `Baboon` and `Tiffany` as training data, we construct codebooks of size 8, 16, 32 and 64 for dimension 4 using the generalized Lloyd algorithm [7]. We then compare the encoding speed of EAHP using binary decision tree and EAHP using our algorithm when encoding the `Lena` image. Excluding I/O access time, we see improvement of 9.83%, 5.67%, 6.65% and 4.47% respectively for the four codebook sizes. First, notice that the improvement for VQ is not as drastic as SQ. This is expected, since we are speeding up only the initial search for closest codevector mean, and the VQ encoding algorithm needs to perform other tasks like computing distortion between input vector and potential candidate vectors. Second, notice that as the size of the codebook increases, the percentage improvement decreases. The reason is that EAHP is increasingly ineffective in ruling out candidate codevectors as the codebook size grows. The bulk of the computation then becomes the computations of distortion between input vector and candidate vectors, and the speed improvement of initial search for closest codevector mean is diminished in the overall picture.

6 Conclusion

In this paper, we seek algorithmic computational optimization of scalar and vector quantizer encoders by exploiting the memory hierarchy of the underlying processor. In particular, by instantiating the dynamic memory model based optimization framework, we find the optimal computation-memory tradeoffs to minimize encoding time of scalar and vector quantizer. In the results section, we see improved performance in both the SQ and VQ case.

A Lloyd's Algorithm

Suppose we choose the distortion metric to be $\mathcal{L}^2$. So given the center of the partition i, C_i, the distortion of this partition is $\int_{\tau_i}^{\tau_{i+1}} p(x)(x - C_i)^2 dx$. The optimal center C_i^* is the center value that minimizes this integral:

$$C_i^* = \arg\min_{\forall C_i} \int_{\tau_i}^{\tau_{i+1}} p(x)(x - C_i)^2 \; dx \tag{22}$$

Taking the first derivative with respect to C_i and equating it to 0, we can find the closed form solution for C_i^*:

$$\begin{aligned}
\frac{d(.)}{dC_i} &= \int_{\tau_i}^{\tau_{i+1}} p(x) \left[\frac{d}{dC_i} (x - C_i)^2 \right] \; dx & (23)\\
&= \int_{\tau_i}^{\tau_{i+1}} p(x)(x - C_i)(-1) \; dx & (24)\\
&= C_i \int_{\tau_i}^{\tau_{i+1}} p(x) \; dx - \int_{\tau_i}^{\tau_{i+1}} p(x) x \; dx & (25)\\
C_i^* &= \frac{\int_{\tau_i}^{\tau_{i+1}} p(x) x \; dx}{\int_{\tau_i}^{\tau_{i+1}} p(x) \; dx} & (26)
\end{aligned}$$

τ_i is the threshold value so that input value x just to the left of it will be quantized to partition $i-1$ with quantized value C_{i-1}, and value just to the right of it will be quantized to partition i with quantized value C_i. Optimal threshold τ_i^* is the value that minimizes these effects of quantization:

$$\tau_i^* = \arg\min_{\forall \tau_i} \left\{ \int_{C_{i-1}}^{\tau_i} p(x)(x - C_{i-1})^2 \; dx + \int_{\tau_i}^{C_i} p(x)(x - C_i)^2 \; dx \right\} \tag{27}$$

We can similarly solve for the closed form of τ_i^* by taking the derivative and equating it to 0:

$$\begin{aligned}
\frac{d(.)}{d\tau_i} &= p(x)(x - C_{i-1})^2 \Big|_{\tau=\tau_i} - p(x)(x - C_i)^2 \Big|_{\tau=\tau_i} & (28)\\
(\tau_i^* - C_{i-1})^2 &= (\tau_i^* - C_i)^2 & (29)\\
\tau_i &= \frac{C_i^2 - C_{i-1}^2}{2(C_i - C_{i-1})} & (30)
\end{aligned}$$

References

[1] K.Lengwehasatit, A.Ortega, "Distortion/Decoding Time Tradeoffs in software DCT-based Image Coding," *ICASSP 97*, 1997.

[2] V. Goyal and M. Vetterli, "Computation-Distortion Characteristics of Block Transform Coding," *ICIP 97*, pp.2729-2732, 1997.

[3] G.Cheung, S.McCanne, C.Papadimitriou, "Software Synthesis of Variable-length Code Decoder using a Mixture of Programmed Logic and Table Lookups," *DCC 99*, March, 1999.

[4] G.Cheung, S.McCanne, "Dynamic Memory Model based Framework for Optimization of IP Address Lookup Algorithms," *ICNP 99*, November, 1999.

[5] G.Cheung, S.McCanne, "An Attribute Grammar Based Framework for Machine Dependent Computational Optimizations of Media Processing Algorithms," *ICIP 99*, October, 1999.

[6] Y.Shoham and A.Gersho, "Efficient Bit Allocation for an Arbitrary Set of Quantizers," *IEEE Trans. ASSP*, vol.36, pp.1445-1453, September 1988.

[7] A.Gersho, R.Gray, *Vector Quantization and Signal Compression*, Kluwer, 1992.

[8] L.Guan, M.Kamel, "Equal-average Hyperplane Partition Method for Vector Quantization of Image Data," Pattern Recognition Letter 13 (1992) 693-699.

[9] T.Cormen, C.Leiserson, R.Rivest, *Introduction to Algorithms*, MIT Press, 1990.