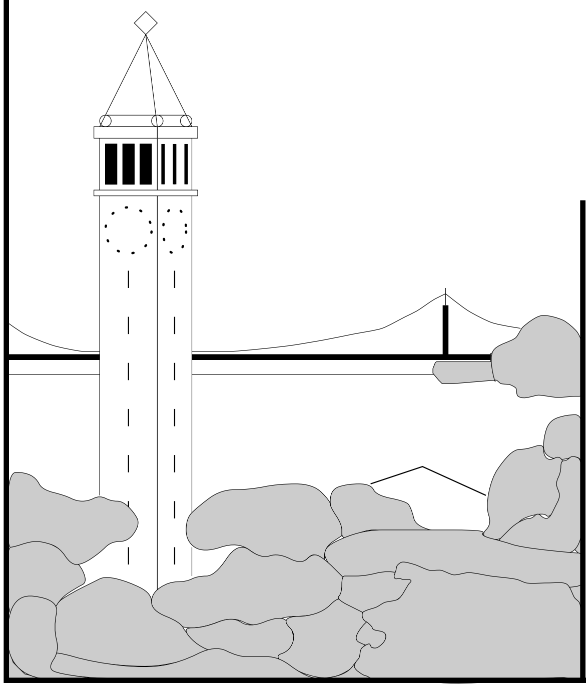
Software Synthesis of Variable-length Code Decoder using a Mixture of Programmed Logic and Table Lookups

Gene Cheung, Steve McCanne, Christos Papadimitriou



Report No. UCB/CSD-99-1040

February 1999

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

Work supported by Grant
442427-25600-3(EB61)

Abstract

Implementation of variable-length code (VLC) decoders can involve a tradeoff between decoding time and memory usage. In this paper, we proposed a novel scheme for optimizing this tradeoff using a machine model abstracted from general purpose processors with hierarchical memories. We formulate the VLC decode problem as an optimization problem where the objective is to minimize the average decoding time. After showing that the problem is NP-complete, we present a Lagrangian algorithm that finds an approximate solution with bounded error. In the resulting framework, an implementation is automatically synthesized by a code generator. To demonstrate the efficacy of our approach, we conducted experiments of decoding codebooks for pruned tree-structured vector quantizer and H.263 motion vector that show a performance gain of our proposed algorithm over single table lookup implementation and logic implementation.

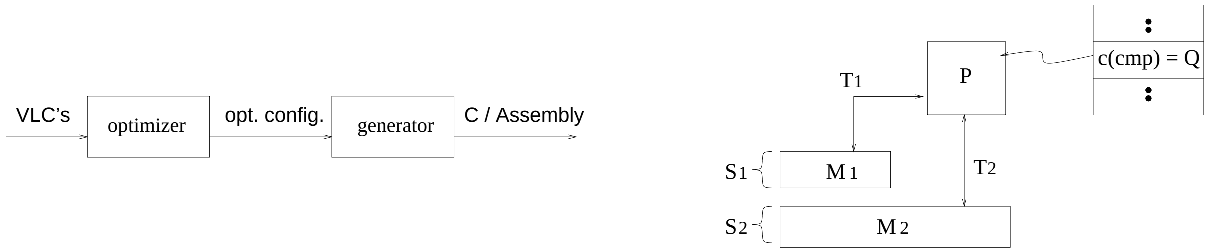
1 Introduction

Many signal processing applications rely upon variable-length coding to reduce distortion and/or encoding bit rate. For example, Huffman code [1] maps a sequence of recurring, statistically independent symbols into a minimally described bit sequence. Likewise, a pruned tree-structured vector quantization (PTSVQ) [2] maps an input vector into one of a finite number of codewords in a codebook via multi-stage approximation. In both cases, the encoder maps each input symbol to a variable-length code (VLC); VLCs are then concatenated to form a bit stream.

Implementation of VLC decoder is often a tradeoff between decoding speed and memory usage. Efficient representations of a set of VLCs often mean a slow decoding process, while fast VLC decoding algorithms usually require large memory space. For example, a simple and efficient representation of a set of VLCs is a binary tree. An associated decoding algorithm starts at the root node, extracts a bit from the input, and follows the corresponding branch to the next node. The process repeats until a leaf is reached, which in turn indicates the symbol. The complexity of this algorithm is $O(d)$, where d is the depth of the tree. If the depth d is large, then the decoding speed of this implementation is slow.

Rather than processing only one bit at each step, a more time-efficient approach is to decode bits in parallel using a lookup table. The decoding algorithm first extracts d bits of the input to form an index into a lookup table. The indexed element contains the symbol that corresponds to the first codeword of the indexed d bits. The complexity per codeword is $O(1)$. Unfortunately, this procedure requires 2^d array elements to be stored in memory. If the depth d is large, then the memory space required for this implementation is large.

Depending on cost criteria, numerous approaches have been offered in the literature. To minimize memory required to represent the Huffman tree, [4, 3] present compact data structures used



a) Optimizer-Generator Pair b) Machine Model

Figure 1: Block Diagram of Proposed System and Machine Model

to store the VLC set while maintaining reasonable decoding speed. To improve the decode speed, [5] proposes to decode groups of n bits each at a time using different context-dependent decoding tables. The price of the speedup is the increase in memory usage for the tables. To avoid excess memory usage, [6] first defines a metric called *memory efficiency*, then presents a tree clustering algorithm that creates data structures with high memory efficiency used for Huffman decoding. To decode very large data symbol set ($n = 10^6$), [7] uses a special set of VLCs called a *canonical code* to implement minimum redundancy coding. Because of its numerical sequence property, a canonical code can be represented without explicitly specifying the binary tree. A fast decoding algorithm was derived based on this property.

Unlike these previous approaches, the goal of our paper is to automatically synthesize an efficient software implementation of a VLC decoder tailored for a machine processor with hierarchical memories. By efficient, we mean a decoder with fast average decoding speed per codeword. Our motivation for this objective is twofold. First, many modern desktop computers employ general purpose processors with hierarchical memories – smaller, faster memory is located close to the processor for fast data access, while larger, slower memory is located further away, storing less frequently used data. Second, when a thin client in a network of computers requests an optimal decoder, our proposed system can generate a decoder tailored to a client's specific processor and send it over the network for immediate use.

In this paper we present an optimizing code generator for VLC. The proposed system, shown in Figure 1a, is divided into two parts. The *optimizer* receives a set of VLCs as input, and outputs a description of the optimal implementation. Optimality is defined with respect to a machine model that captures the memory size and access speed of different hierarchical memories of a processor. The *generator* translates the description into a mixture of C and in-line Assembly code, which performs a mixture of table-lookups and programmed logical comparisons. Note that we are solving the general case of decoding VLC, a superset that includes the set of minimum redundancy code. While minimum redundancy code has the freedom to choose any codebook that has the minimum average codeword length, the general case assumes the particular choice of codebook is important. Applications where the codebook is important include PTSVQ, alphabetic minimum redundancy codes [8] etc.

The outline of the paper is as follows. In section 2, the proposed machine model is discussed, and the optimization problem is formalized. In section 3, we show the optimization problem is NP-complete. In section 4, we present an approximate solution to the optimization problem. In section 5, implementation of the code generator is discussed, and results are presented. Finally, a conclusion is provided in section 6.

2 Machine Model

Modern general-purpose processors use hierarchical memories to enhance performance, where small, fast memories are located near the CPU and larger, slower (and cheaper) memories are situated further away. Consequently, the execution speed of a machine instruction that accesses memory depends on the type of memory referenced. A machine model that reflects this characteristic is shown in Figure 1b. If the processor P accesses a datum residing in type 1 memory M_1 (type 2 memory 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. In Figure 1b, the cost of a logical comparison *cmp* is Q. For the chosen machine model, the size of the type 1 memory is S_1, and the size of the type 2 memory is $S_2 = \infty$.

Given such a machine model, we can evaluate the average decoding time of a VLC decoding algorithm that uses a mixture of lookup tables and programmed logics. For example, if the set of VLCs in Figure 2a is implemented as shown in Figure 2b, where a width 2-bit lookup table located in type 1 memory is first indexed, followed possibly by a logical comparison, then the average lookup time is: $p_0(Q+T_1)+p_1(Q+T_1)+p_2T_1+p_3T_1+p_4T_1$, where p_j is the probability of symbol j. We call a particular arrangement of logical comparisons and lookup tables in hierarchical memories a *configuration*. More generally, we can compute the average decoding time per symbol of a configuration b, denoted $H(b)$, as:

$$H(b) = \sum_j p_j(a_jT_1 + b_jT_2 + c_jQ) \tag{1}$$

a) binary tree representation b) Hybrid of lookup table and logic

Figure 2: Example of a Configuration

where a_j (b_j) is the number of type 1 (type 2) memory access needed to decode symbol j, and c_j is the number of logical comparison needed. In other words, a_j (b_j) is the number of lookup tables residing in type 1 (type 2) memory used in the decoding process of symbol j.

Armed with this model, we can pose a well-formed optimization problem that we call the "VLC decode problem":

$$\min_{b \in B} H(b) \qquad \text{s.t.} \qquad R(b) \leq S_1 \tag{2}$$

where B is the set of possible configurations and $R(b)$ is the total size of lookup tables assigned to type 1 memory. In words, the problem is: given a set of VLCs and their associated probabilities, what is the optimal configuration such that the decoding time is minimized? Note that the optimal configuration must not assign tables to type 1 memory in such a way that it exceeds the memory capacity of the machine model. This is of real concern in practice, where the length of the longest codeword can be 13 bits or longer [9]; a full lookup table containing 2^{13} elements would be too large to fit into type 1 memory — L1 cache — of common processors.

We can also write the cost of a configuration in terms of the probability density (weight) of nodes in the binary tree. For example, the decoding time of the configuration shown in Figure 2b can be written as:

$$H(b) = w_4 T_1 + w_1 Q \tag{3}$$

where $w_4 = p_0 + \cdots + p_4$ is the *weight* of node 4, and $w_1 = p_0 + p_1$ is the weight of node 1. Cost of a configuration written in this form is used in section 4.

In the next section, we will show that even in the case when we use only lookup tables, the problem is NP-complete. Therefore the general problem using a combination of logical comparisons and lookup tables is also NP-complete, and we turn to an approximate solution, which we present in section 4.

	x_2	x_1	x_0	y_2	y_1	y_0	z_2	z_1	z_0
a_0	1	0	0	1	0	0	1	0	0
a_1	0	0	1	0	0	1	0	0	1
a_2	0	0	1	1	0	0	0	1	0
a_3	0	1	0	0	1	0	0	1	0
K	1	1	1	1	1	1	1	1	1

Figure 3: Partial Sum Version of 3D Matching Problem: $N = 3$, $M = 4$

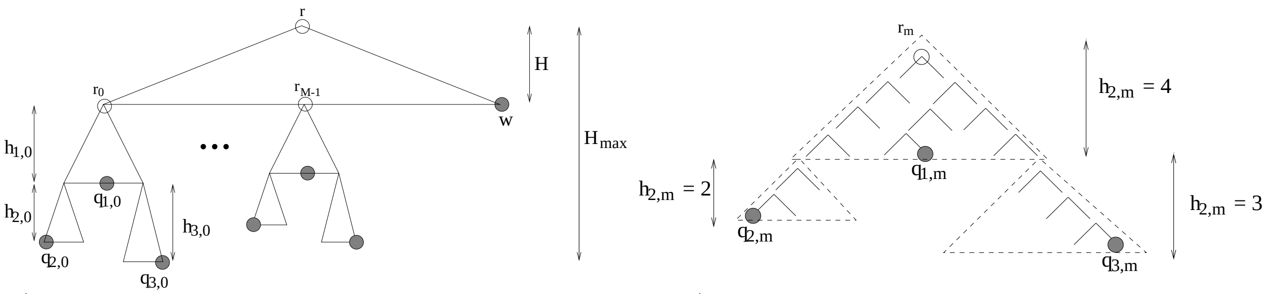
3 NP-Completeness Proof

We first rephrase the VLC decode problem as a decision problem: given a set of VLCs with associated probabilities, does there exist a configuration of lookup tables and table assignments to hierarchical memories that has a cost below a target cost $\bar{C}$, where cost is expressed in (1)? In this section, we present the proof of NP-completeness for this decision problem.

3.1 3D Matching Problem

The proof is by reduction from a version of the "3D matching problem" [10]. This well-known NP-complete problem assumes the input is categorized into three distinct groups, say men, women and pets, each of size N. A list of 3-tuples of size $M > N$ specifies all possible matches of men, women and pets. For example, a tuple (i_m, j_m, k_m) specifies man i_m, woman j_m and pet k_m is a possible match. The decision problem is: given a list of 3-tuples, is it possible to select N of M possible matches such that each of N men, women and pets is uniquely assigned to one match.

The same problem can be reformulated as the "partial sum" version as follows. Suppose we have M numbers in numeric base $M+1$, each with $3N$ digits. We transform each 3-tuple (i_m, j_m, k_m) in the input list to a number $a_m = (M+1)^{2N+i_m} + (M+1)^{N+j_m} + (M+1)^{k_m}$, $\forall i_m, j_m, k_m \in \{0...N-1\}$. Notice each number has exactly three 1's in the three digit positions and the rest of the digits are zeros, and further that overflow in any digit position is avoided for any subset of numbers by selecting the numeric base to be $M+1$. Now, the decision problem is: does there exist a subset of these M numbers such that the sum of the subset is exactly K, a number with ones in all $3N$ digit positions. An example of the partial sum version is shown in Figure 3 for $N = 3$ and $M = 4$. We see that the numbers in the subset $\{a_0, a_1, a_3\}$ add up to K. This version of the 3D matching problem is equivalent to the original version discussed earlier.



a) Construction Overview b) Gadget Construction for each a_m

Figure 4: Proof Constructs used in the NP-Complete Proof of VLC decode Problem

3.2 Overview of Proof

By reduction from the partial sum version of the 3D matching problem, we will prove the VLC decode problem is NP-complete, stated below as a theorem.

Theorem 1 *The VLC decode problem using only lookup tables and under one hierarchical memory constraint is NP-complete.*

We now sketch the outline of the proof. For every instance of the partial sum version of the 3D matching problem, we create a corresponding instance of the VLC decode problem, polynomially transformed from the instance of the partial sum problem. If we solve the corresponding instance of the VLC decode decision problem, we also solve the original instance of the partial sum decision problem, and therefore the VLC decode problem is at least as hard as the partial sum problem. Since the partial sum problem is NP-complete, the VLC decode problem is also NP-complete.

We construct the corresponding instance of the VLC decode problem as follows. We first construct the set of VLCs, represented by a binary tree. It is a full binary tree with root r of height H such that $2^H > M$, attached at the bottom with M subtrees — one for each number a_m, $m \in \{0 \ldots M-1\}$. This is shown in Figure 4a. In addition, there is one non-zero probability leaf at the bottom of the full tree, called the *heavy leaf*, with probability w. The subtrees are the *gadgets* necessary to map the numbers a_m's in the 3D matching problem to the VLC decode problem. Each subtree m is a concatenation of three mini-trees of height $h_{1,m}$, $h_{2,m}$ and $h_{3,m}$ and has a single leaf with non-zero probability $q_{1,m}$, $q_{2,m}$ and $q_{3,m}$ respectively. Mini-tree 2 and 3 are single sided, and mini-tree 1 has three branches of the same height $h_{1,m}$, with non-zero probability leaf in the middle branch and concatenations to tree 2 and 3 at the other branches. See Figure 4b for an example of the three mini-trees in a subtree m.

We first set type 1 memory size S_1 of the machine model to be $2^H + K$. We select H so that a lookup table of width $h > H$ will not fit in type 1 memory ($2^{H+1} > S_1$). Now we can set the probability of the heavy leaf w large enough so that the optimal configuration must contain a type 1 memory assigned lookup table rooted at r of width H — this is the only way to ensure that decoding the codeword corresponding to the heavy leaf takes only one type 1 memory access. This leaves K type 1 memory space for the M subtrees. Knowing the optimal configuration must contain the above mentioned table at root r, the decision problem is now reduced to: does there exist a set of configurations for the M subtrees such that the resulting cost is smaller than $\bar{C}$, where K is the size of the type 1 memory available for the subtrees?

We call the subtree configuration that employs one type 1 memory assigned lookup table for each of the three mini-trees as the *3-triangle configuration*. We select $h_{1,m}$, $h_{2,m}$ and $h_{3,m}$ properly so that the combined size of the three tables of subtree m is a_m. Therefore the type 1 memory usage for 3-triangle configured subtree m is also a_m. We call another subtree configuration that uses one type 2 memory assigned lookup table for the entire subtree m the *default configuration*. The type 1 memory usage in this case is 0. By selecting machine model parameter T_1, T_2 and leaf probabilities $q_{1,m}$, $q_{2,m}$, $q_{3,m}$ properly, 3-triangle configuration of subtree m reduces lookup cost by a_m over the default configuration. Moreover, other configurations besides 3-triangle and default configuration are inferior in that they use up too much type 1 memory space while reducing cost only marginally.

Given the above constructions, we claim the following: If there exists a subset of numbers that adds up to K in the partial sum problem, then there exists a corresponding subset of subtrees in the VLC decode problem, in 3-triangle configurations, that will reduce the cost by K while using up exactly K leftover type 1 memory. The converse is also true. That means by answering the corresponding VLC decode decision problem, we also answer the partial sum decision problem. Therefore the VLC decision problem is as least as hard as the partial sum problem, and so the VLC decode problem is NP-complete.

The parameters of the VLC decode problem must be carefully set in relations to the parameters of the 3D matching problem. Figure 5 shows the dependencies between parameters of the two problems. We first modify the partial sum version of the 3D match problem slightly by selecting the numeric base of the M numbers to be 2^B, where B is:

$$B = \lceil \log(M + 1) \rceil \tag{4}$$

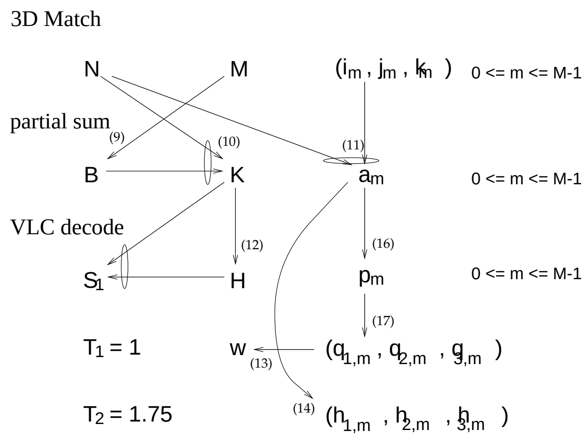


Figure 5: Dependencies between Parameters of 3D Match and VLC Decode

Note that $2^B \geq M+1$ and there is still no overflow in any digit position. The target partial sum K and the number a_m that corresponds to a possible match (i_m, j_m, k_m) can now be written as:

$$K = \sum_{n=0}^{3N-1} 2^{Bn} \tag{5}$$

$$a_m = 2^{B(2N+i_m)} + 2^{B(N+j_m)} + 2^{B(k_m)} \tag{6}$$

We now set the parameters of the VLC decode problem. First, we need to select the height H of the full tree rooted at r to be large enough such that M subtrees can be attached to the bottom, i.e. $2^H \geq M$. We accomplish this by setting H as:

$$H = \lfloor \log K \rfloor + 1. \tag{7}$$

Second, in order to guarantee the optimal configuration contains the table rooted at r of height H in type 1 memory, heavy leaf located at level H from root of the tree will have a large probability w:

$$w = 1.5 \sum_{m=0}^{M-1} \sum_{i=1}^{3} q_{i,m} \tag{8}$$

The heavy leaf has large enough probability to influence the optimal solution to contain a height H table rooted at r in type 1 memory, so that accessing this leaf will take only one fast memory access, T_1.

Third, in order to have the size of 3-triangle configured subtree m be a_m, we need to select the height of the three associated trees to be:

$$(h_{1,m}, h_{2,m}, h_{3,m}) = (B(2N + 1 + i_m), B(1 + k_m), B(N + 1 + j_m)) \tag{9}$$

where (i_m, j_m, k_m) is the corresponding 3-tuple that states a possible match for man i_m, woman j_m and pet k_m.

Finally, we need to select the leaf probabilities $q_{1,m}$, $q_{2,m}$ and $q_{3,m}$, and memory access time T_1 and T_2 to satisfy two conditions: i) the improvement of 3-triangle configuration of subtree m over default configuration is exactly a_m; and, ii) the 3-triangle configuration is in some sense more "desirable" than any other configuration. With these goals in mind, we set these parameters as follows:

$$\begin{aligned}
(T_1, T_2) &= (1, 1.75) && (10)\\
p_m &= (4/17)a_m && (11)\\
(q_{1,m}, q_{2,m}, q_{3,m}) &= (10p_m, 7p_m, 6p_m) && (12)
\end{aligned}$$

3.3 Details of Proof

We now show the details of the proof. We begin by showing the optimal solution must include a lookup table rooted at r of height H assigned to type 1 memory.

Lemma 1 *A lookup table rooted at r of height H, assigned to type 1 memory, must be part of the optimal configuration.*

Proof 1

We divide the proof into two cases: i) showing that lookup table rooted at r of height $h > H$ is sub-optimal, and ii) showing that lookup table rooted at r of height $h < H$ is also sub-optimal. For the first case, we notice that table of height $h \geq H + 1$ will not fit into type 1 memory:

$$\begin{aligned}
2^H &= 2^{\lfloor \log K \rfloor + 1}\\
&> 2^{\log K} = K\\
2^{H+1} &> 2^H + K = S_1 && (13)
\end{aligned}$$

The optimal configuration among choices of tables $h \geq H+1$ is to use table of height $H_{\max}$ and put it in type 2 memory. Because the root table is in type 2 memory, accessing each codeword of each leaf must take at least one slow memory access, T_2. Given that is the case, we make that memory access the only one needed by using the largest possible lookup table, thereby minimizing the cost. The cost of the configuration, C', is:

$$C' = wT_2 + T_2 \sum_{m=0}^{M-1} \sum_{i=1}^{3} q_{i,m} \tag{14}$$

We now find an upper bound cost of an optimal configuration using table rooted at r of height H. A conservative configuration puts the height H table in type 1 memory and puts all the M default configured subtrees in type 2 memory. Note that only 2^H type 1 memory is used, with K memory space left empty. The cost of this choice is:

$$C = wT_1 + (T_1 + T_2) \sum_{m=0}^{M-1} \sum_{i=1}^{3} q_{i,m} \tag{15}$$

Comparing the two cost:

$$\begin{aligned} C' - C &= (T_2 - T_1)w - T_1 \sum_{m=0}^{M-1} \sum_{i=1}^{3} q_{i,m} \\ &= (0.75)1.5 \sum_{m=0}^{M-1} \sum_{i=1}^{3} q_{i,m} - \sum_{m=0}^{M-1} \sum_{i=1}^{3} q_{i,m} > 0 \end{aligned} \tag{16}$$

We show that the upper bound cost of an optimal configuration using table rooted at r of height H is lower than the lower bound cost of a configuration using table of height $h > H$. Therefore, we conclude that configurations using table rooted at r of height $h > H$ is sub-optimal.

Consider now the case of using lookup table rooted at r of height $h < H$. We know that the table must be assigned to type 1 memory, for same reason discussed previously. Because the table has height smaller than H, it will take at least $2T_1$ to access the heavy leaf and any leaves on the M subtrees. The lower bound cost is:

$$C'' = w2T_1 + 2T_1 \sum_{m=0}^{M-1} \sum_{i=1}^{3} q_{i,m} \tag{17}$$

Similar calculation will show the upper bound cost of configurations using table rooted at r of height H is smaller than C''. Therefore, we conclude that the table of height $h < H$ is sub-optimal.

We have shown that table of height $h > H$ and of height $h < H$ are both more costly than table of height H. Therefore Lemma 1 is proven. □

Given the optimal solution contains table rooted at r of height H assigned to type 1 memory, we have simplified the problem to allocating M subtrees in K leftover type 1 memory to minimize cost.

We now prove that with the parameters chosen, 3-triangle configuration of a subtree has the two properties we discussed above. The following lemma proves the first property.

Lemma 2 *Employing 3-triangle configuration of subtree m has cost reduction a_m over default configuration of subtree m.*

Proof 2

3-triangle configuration of subtree m has the following cost (beyond lookup cost of table rooted at r):

$$\begin{aligned} C &= q_{1,m}T_1 + q_{1,m}2T_1 + q_{1,m}2T_1 \\ &= 10p_m + (7p_m)2 + (6p_m)2 = 36p_m \end{aligned} \tag{18}$$

The cost of default configuration, denoted as the default cost, is shown below:

$$\begin{aligned} C_d &= q_{1,m}T_2 + q_{2,m}T_2 + q_{3,m}T_2 \\ &= 23p_m(1.75) = 40.25p_m \end{aligned} \tag{19}$$

Comparing these two costs:

$$C_d - C = 4.25(4/17 * a_m) = a_m \tag{20}$$

Therefore the reduction in cost of 3-triangle configuration of subtree m over default configuration is a_m. □

Before we show how the 3-triangle configuration of a subtree is more "desirable" than other configurations, we first prove some configurations are intrinsically locally sub-optimal, and therefore can be removed from consideration when constructing the optimal subtree configuration. They are configurations that has lookup tables rooted at r_m and height $h \leq h_{1,m}$, and different from the 3-triangle configuration. They are discussed in the next two lemmas.

Lemma 3 *A subtree configuration that contains a lookup table rooted at r_m of height $h < h_{1,m}$ is locally sub-optimal.*

Proof 3

We first find the lower bound cost of these configurations. If a subtree configuration contains a lookup table rooted at r_m of height $h < h_{1,m}$, then to access the single leaf on tree 1, as well as leaves on tree 2 and 3, will take at least two fast memory load. The lower bound cost C' is:

$$\begin{aligned} C' &= (q_{1,m} + q_{2,m} + q_{3,m})2T_1 \\ &= (10p_m + 7p_m + 6p_m)2 = 46p_m \end{aligned} \tag{21}$$

Recall the default cost C_d is $40.25p_m$. Since the default cost is lower than the lower bound cost of these configurations, we can lower the cost by removing this subtree from type 1 memory and placing it in type 2 memory. Therefore this configuration is locally sub-optimal. □

The next lemma deals with configurations using lookup table rooted at r_m of height $h = h_{1,m}$, but different from the 3-triangle configuration.

Lemma 4 *Suppose a subtree configuration, different from the 3-triangle configuration, also has a lookup table rooted at r_m of height $h = h_{1,m}$. This configuration is locally sub-optimal.*

Proof 4

A subtree configuration with lookup table rooted at r_m of height $h = h_{1,m}$ can differ from the 3-triangle configuration in two ways: i) by placing the single lookup table of tree 1, 2 or 3 in type 2 memory instead of type 1; and, ii) by using more than one lookup table in tree 2 and/or tree 3. Simple calculation will show that by placing any single lookup table in tree 1, 2 or 3 will result in cost larger than the default cost. Similarly, if one create more than one lookup table in tree 2 and/or tree 3, the number of memory access required for leaf on tree 2 and/or 3 will increase, and the resulting cost is also larger than the default cost. Therefore, a subtree configuration with lookup table rooted at r_m of height $h = h_{1,m}$, if different from the 3-triangle configuration, is locally sub-optimal. □

We now show that the remaining configurations are less "desirable" compare to 3-triangle configuration. By desirable, we mean it has a high *performance-price ratio*, defined as the ratio of cost reduction of a configuration over the default configuration, to the amount of type 1 memory space required for the configuration. For example, if a configuration y has cost $C(y)$ and requires type 1 memory space $S(y)$, then the performance-price ratio $r(y)$ for this configuration is:

$$r(y) = \frac{C_d - C(y)}{S(y)} \tag{22}$$

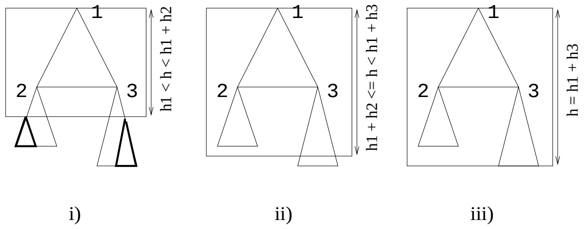


Figure 6: Three Cases for Proof of Optimal Performance-Price Ratio of 3-Triangle Configuration

The following lemma shows the 3-triangle configuration has the best possible performance-price ratio.

Lemma 5 *The 3-triangle configuration of a subtree m has strictly higher performance-price ratio than other configurations.*

Proof 5

We will divide the proof into three cases: i) configurations with lookup table rooted at r_m of height $h_1 + 1 \le h < h_1 + h_2$, ii) $h_1 + h_2 \le h < h_1 + h_3$, and iii) $h = h_1 + h_3$. Recall from Lemma 3 and 4 that configurations with table rooted at r_m of height $h \le h_1$, if differ from 3-triangle configuration, are locally suboptimal. As a result, the cost improvement over default configuration is negative, and their performance-price ratios are also negative.

The performance-price ratio of 3-triangle configuration x, denoted as $r(x)$, is computed as follows:

$$r(x) = \frac{40.25p_m - 36p_m}{2^{h_{1,m}} + 2^{h_{2,m}} + 2^{h_{3,m}}} = \frac{4.25p_m}{2^{h_{1,m}} + 2^{h_{2,m}} + 2^{h_{3,m}}} \tag{23}$$

We will show in all three cases, the ratio will be strictly smaller than $r(x)$.

1. $h_1 + 1 \le h < h_1 + h_2$

   Configurations using lookup table rooted at r_m of this height means that leaf on tree 1 need one type 1 memory access, and leaves on tree 2 and 3 need two type 1 memory access. Note that neither the leaf on tree 2 or tree 3 can require more than two fast memory access, or the cost will exceed the default cost. That has two consequences: i) the configuration must employ one lookup table for each of the remaining tree 2 and 3, as shown in Figure 6i; and, ii) the configuration has a cost equal to the cost of 3-triangle configuration. This means if the size of this configuration is larger than 3-triangle configuration, then this configuration will have a smaller performance-price ratio than the 3-triangle configuration. The size can be lower bounded as follows:

$$\begin{aligned} S(y_1) &= 2^h + 2^{h_{1,m}+h_{2,m}-h} + 2^{h_{1,m}+h_{3,m}-h} \\ &> 2^{h_{1,m}+1} > S(x) \end{aligned} \tag{24}$$

Since $S(y_1) > S(x)$, we conclude that configurations with lookup table rooted at r_m and height $h_1 + 1 \leq h < h_1 + h_2$ has strictly smaller performance-price ratio than 3-triangle configuration.

2. $h_1 + h_2 \leq h < h_1 + h_3$

We find the upper bound of performance-price ratio for these configurations as follows. Leaf at tree 3 must take a minimum of two fast memory access, so the minimum cost can be computed as:

$$C(y_2) = [q_{1,m} + q_{2,m} + 2 * q_{3,m}]T_1 = 29p_m \tag{25}$$

Since the table rooted at r_m must be in type 1 memory, the size is at least $2^{h_{1,m}+h_{2,m}}$. The upper bound of the performance-price ratio is:

$$r(y_2) \leq \frac{(40.25 - 29)p_m}{2^{h_{1,m}+h_{2,m}}} = \frac{11.25p_m}{2^{h_{1,m}+h_{2,m}}} \tag{26}$$

We would like to find the smallest value $h_{2,m}$ can take on. Note that the 3D matching problem is non-trivial when the number of members in a group $N \geq 2$. This means $M \geq 2+1$, $B \geq 2$, and $h_{2,m} \geq 2$.

$$r(y_2) \leq \frac{11.25p_m}{2^{h_{1,m}+2}} = \frac{2.8125p_m}{2^{h_{1,m}}} \tag{27}$$

We need to change the denominator of $r(x)$ to $2^{h_{1,m}}$ to have a meaningful comparison. We first note that $B \geq 2$ implies $h_{1,m} \geq h_{3,m} + 2$. The following bound follows:

$$1/2 * 2^{h_{1,m}} > 2^{h_{2,m}} + 2^{h_{3,m}} \tag{28}$$

We can now lower bound $r(x)$ as follows:

$$\begin{aligned} 2^{h_{1,m}} + 2^{h_{2,m}} + 2^{h_{3,m}} &< 2^{h_{1,m}} + 1/2 * 2^{h_{1,m}} = 2^{h_{1,m}}(1 + 1/2) \\ r(x) &\geq \frac{4.25p_m}{2^{h_{1,m}}3/2} \geq \frac{2.833p_m}{2^{h_{1,m}}} \end{aligned} \tag{29}$$

We see that the lower bound of $r(x)$ is larger than the upper bound of $r(y_2)$. Therefore the performance-price ratios for this set of configurations are smaller than the 3-triangle configuration.

3. $h = h_1 + h_3$

   Applying similar bounding techniques to previous case, we arrive at the following upper bound of the performance-price ratio of these configurations y_3 as:

$$r(y_3) = \frac{40.25p_m - 23p_m}{2^{h_{1,m}+h_{3,m}}} \leq \frac{17.25p_m}{2^{h_{1,m}+6}} \leq \frac{0.270p_m}{2^{h_{1,m}}} \tag{30}$$

   Therefore we can conclude $r(y_3) < r(x)$.

We have shown that the performance-price ratio for all three sets of configurations are smaller than the ratio for 3-triangle configuration. Therefore the lemma is proven. □

Armed with the above lemmas, we are ready to proof Theorem 1.

Proof 1

We need to prove the theorem in both direction. First, we will prove that if there exists a subset of numbers in the partial sum version of the 3D matching problem that adds up to K, then the reduction in cost will also be K.

Suppose there exists a subset of numbers that adds up to K. We know that lookup table rooted at r of height H must be in type 1 memory by lemma 1. That leaves K memory space left for M subtrees. For each subtree m that corresponds to a number a_m in the subset, we use 3-triangle configuration — each has type 1 memory usage a_m (by construction) and reduction in cost a_m (by lemma 2). For all other subtree, we use default configuration — each has type 1 memory usage 0 (by construction) and reduction in cost 0 (by definition). The total type 1 memory usage of the subtree will be K, while the total reduction in cost is also K. This proves the first part.

Suppose there is a reduction in cost of K in the VLC decode problem. Again, lookup table rooted at r of height H must be in type 1 memory by lemma 1, leaving K memory space left for M subtrees. To achieve a reduction of exactly K, we claim that there must be a subset of subtrees in 3-triangle configurations that fit exactly in the K leftover memory space, thereby reducing the cost by K. It cannot be done any other way. The reason is the following: all 3-triangle configurations of all subtrees has the strictly largest improvement-price ratio, 1. If one attempts to reduce the cost by any other configurations, one will do so at a lower ratio, resulting in a less-than-K improvement. This proves the second part.

We have proved the theorem in both direction, and so the theorem is proven. □

4 Lagrange Approximation Algorithm

Given that the VLC decode problem is NP-complete, we propose the following approximate algorithm that has fast execution time and terminates with bounded error. We first present a high-level description of the algorithm, then we detail the notion of *singular value* — special multiplier values used in the algorithm to ensure it converges in finite time.

4.1 Development of Algorithm

Our algorithm is based on an application of Lagrange multipliers to discrete optimization problems with constraints, as was done by Shoham and Gersho [11] for bit allocation problems. Instead of solving the original constrained VLC decode problem in (2), we solve the corresponding Lagrangian problem, which is unconstrained:

$$\min_{b \in B} H(b) + \lambda R(b) \tag{31}$$

where λ is a Lagrange multiplier with non-negative value, and $H(b)$, B and $R(b)$ are defined as before. If there exists a multiplier value λ^* such that the solution of the Lagrangian problem, b^*, satisfies the constraint of original problem with equality — i.e. $R(b^*) = S_1$, then b^* is also the solution to the original problem. Because the Lagrangian is unconstrained, it is potentially easier to solve. However, there is an additional step of adjusting the multiplier value λ so that the constraint variable, $R(b)$, satisfying the constraint in (2).

To solve (31) for a particular value of λ, we first represent the set of VLCs in question by a binary tree, where nodes are numbered in post-order with root r. We define a function $f_\lambda(i)$, which returns the minimum Lagrangian cost, $H(b) + \lambda R(b)$, of all possible configurations for the binary tree rooted at node i for given multiplier value λ. We can solve $f_\lambda(i)$ via the following case analysis. At node i, we have three choices: i) perform a logical comparison at node i with cost $w_i Q$; ii) create a lookup table at node i of some width h and place it in type 1 memory with cost $w_i T_1 + \lambda 2^h$; and, iii) create a table of width h and place it in type 2 memory with cost $w_i T_2$. The minimum of these three costs for all possible table width plus the recursive cost of the children nodes will be the cost of the function at node i, expressed below:

$$f_\lambda(i) = \min \left\{ w_i Q + \sum_{j \in L_{1,i}} f_\lambda(j),\ \min_{1 \le h \le H_i} \left[w_i T_1 + \lambda 2^h + \sum_{j \in L_{h,i}} f_\lambda(j) \right],\ \min_{1 \le h \le H_i} \left[w_i T_2 + \sum_{j \in L_{h,i}} f_\lambda(j) \right] \right\} \tag{32}$$

where H_i is the height of binary tree rooted at node i, and $L_{h,i}$ is the set of nodes at height h of tree rooted at node i. We can simplify (32) by the following observations. First, since there is no penalty cost for placement of lookup table in type 2 memory, the best possible choice given a table at node i is assigned to type 2 memory is to create a width H_i table — this eliminates the cost of the children nodes. Second, we can restrict our search space of configurations, B in (2) and (31), to the set of configurations that does not assign a table of size greater than S_1 to type 1 memory — a necessary condition to satisfy constraint in (2). Now we can simplify (32):

$$f_\lambda(i) = \min \left\{ w_i Q + \sum_{j \in L_{1,i}} f_\lambda(j), \min_{1 \leq h \leq \lfloor \log_2 S_1 \rfloor} \left[w_i T_1 + \lambda 2^h + \sum_{j \in L_{h,i}} f_\lambda(j) \right], w_i T_2 \right\} \tag{33}$$

We note that there are overlapping sub-problems when solving $f_\lambda(r)$ using (33); if s is a children node of r and t is a children node of s, then $f_\lambda(t)$ will be used in the calculation of $f_\lambda(r)$ as well as the calculation of $f_\lambda(s)$. To avoid solving the same sub-problem more than once, we use a dynamic programming table $F_\lambda[\,]$ of size $r * 1$ to store the calculated values $f_\lambda(i)$ for $i = 1...r$. Each time the function $f_\lambda(i)$ is called, it first checks if the entry $F_\lambda[i]$ has been filled. If it has, then $f_\lambda(i)$ simply returns the value $F_\lambda[i]$. Otherwise, it calculates the value using (33) and stores it in the table. After solving the Lagrangian problem using (33), we have a configuration, denoted by b^*, that minimizes the Lagrangian problem for a particular multiplier value λ.

The crux of the algorithm is to find λ such that the memory size constraint is met with equality, i.e. $R(b^*) = S_1$. It can be shown that the constraint variable $R(b^*)$ is inverse proportional to the multiplier λ. Therefore, a simple strategy to search for the appropriate multiplier value is to do binary search on the real line to drive $R(b^*)$ to the actual memory size S_1. Note that there may not exist a multiplier value such that $R(b^*) = S_1$. In that case, we find the smallest multiplier value such that $R(b^*) < S_1$. The solution to the Lagrangian now becomes an approximate solution, with the error bounded by the following theorem:

Theorem 2 *Let b^* be the optimal solution to (2). Let b_1, b_2 be optimal solutions to (31) for multipliers λ_1, λ_2 respectively, such that $R(b_1) < S_1$ and $R(b_2) > S_1$. The error of the approximate solution b_1 to (2) can be bounded as follows:*

$$|H(b_1) - H(b^*)| \leq |H(b_1) - H(b_2)| \tag{34}$$

See lemma 3 of [12] for a proof of this theorem.

4.2 Singular Values — multiplier values with multiple solutions

When the constraint variable is close to the memory size, there is a faster method to find the next multiplier value than binary search. Because the problem is discrete, there are only finite number

a) constraint variable vs. multiplier b) Lagrangian cost vs. multiplier

Figure 7: Constraint variable and Lagrangian cost as functions of multiplier

of optimal configurations for $0 \leq \lambda \leq \infty$. As a consequence, if we sweep λ from 0 to ∞, there is a discrete set of multiplier values at which the optimal configuration changes from one to another. In Figure 7a, we see that the constraint variable $R(b^*)$ is a decreasing step function with respect to multiplier λ. Notice at special values of λ, there are multiple optimal configurations, denoted by circles, and therefore multiple values of constraint variable. For example, there are two values of $R(b^*)$, R_2 and R_1, that resulted simultaneously from two optimal configurations for $\lambda = \lambda_2$. These unique values of λ which yield multiple optimal solutions are called *singular values* in [11].

An important observation is that neighboring singular values share a common optimal solution. For example, singular values λ_1 and λ_2 share a common optimal solution with $R(b^*) = R_2$. Because constraint variable $R(b^*)$ is non-increasing with respect to multiplier λ, together with the above observation, we can conclude that by solely looking at the optimal configurations of the singular values, it is sufficient to discover all configurations that are solutions to the Lagrangian. Our approach when constraint variable $R(b^*)$ is close to constraint S_1, is to step to the neighboring multiplier value until the best possible value is found. This approach is similar to the one in [11].

To find the neighboring singular value, we first observe from (32) that by construction, the optimal configuration has Lagrangian cost of form:

$$f_\lambda(i) = \sum_{x \in X} w_x T_1 + \sum_{x \in X} 2^{h_x} \lambda + \sum_{y \in Y} w_y T_2 + \sum_{z \in Z} w_z Q \tag{35}$$

where X is a set of tables assigned to type 1 memory, Y is a set of tables assigned to type 2 memory, and Z is a set of nodes performing logical comparisons. Rewriting the equation yields a simpler representation: a linear function of λ with slope m_i and y-intercept c_i:

$$\begin{aligned} f_\lambda(i) &= c_i + m_i \lambda && (36)\\ c_i &= \sum_{x \in X} w_x T_1 + \sum_{y \in Y} w_y T_2 + \sum_{z \in Z} w_z Q && (37)\\ m_i &= \sum_{x \in X} 2^{h_x} && (38) \end{aligned}$$

Note that this linear function is the optimal solution to the Lagrangian only within a small neighborhood of the current multiplier value λ. As λ increases, if another configuration with a different slope and y-intercept becomes the minimum of all configurations, then that configuration becomes the optimal solution to the Lagrangian. In Figure 7b, as λ increases from $\lambda_1 - \Delta$ to $\lambda_1 + \Delta$, optimal configuration switches from x_1 to x_3. As shown, minimum Lagrangian cost as function of the multiplier, $L(\lambda) = f_\lambda(r)$, is a piecewise linear function. Locating the point at which $L(\lambda)$ switches from one linear piece to another, means locating where the optimal configuration changes, and therefore where constraint variable $R(b^*)$ changes.

To locate the larger neighboring singular value, we first define $g_\lambda(i)$ as a function that returns the next potential larger singular value for the tree rooted at node i. This value can be derived from one of two cases. First, it is the value at which a new configuration that uses a new lookup operator at node i (for example, a logical comparison at node i instead of a type 1 memory lookup table), in combination with the configurations of the children nodes, becomes optimal as the multiplier value increases. Second, it is the value at which one of the children nodes of node i changes its optimal configuration, which affects the optimality calculation for node i. $g_\lambda(i)$ will return the smaller of these two values, as expressed in the following pseudo-code:

1. $temp := I\left([c_i, m_i],\ \ [w_i Q + \sum_{j \in L_{1,i}} c_j,\ \sum_{j \in L_{1,i}} m_j]\right)$
   if $temp > \lambda$, **then** $g_\lambda(i) := temp$ `// check config. w/ logic at node` i
   else $g_\lambda(i) := \infty$
2. $temp := \min_{1 \leq h \leq \lfloor \log S_1 \rfloor} \left\{ I\left([c_i, m_i],\ \ [w_i T_1 + \sum_{j \in L_{h,i}} c_j,\ 2^h + \sum_{j \in L_{h,i}} m_j]\right) \right\}$
   if $temp > \lambda$ **&** $temp < g_\lambda(i)$, **then** $g_\lambda(i) := temp$ `// check config. w/ type 1 memory table at node` i
3. $temp := I([c_i, m_i],\ [w_i T_2, 0])$
   if $temp > \lambda$ **&** $temp < g_\lambda(i)$, **then** $g_\lambda(i) := temp$ `// check config. w/ type 2 memory table at node` i
4. $temp := \min_{j \in L_{1,i}} g_\lambda(j)$
   if $temp > \lambda$ **&** $temp < g_\lambda(i)$, **then** $g_\lambda(i) := temp$ `// check the potential s.v.'s of children nodes`

where function $I([c_1, m_1], [c_2, m_2])$ takes in the slopes m's and y-intercepts c's of two lines, and returns the intersection point. If they are parallel lines, it returns ∞.

$g_\lambda(i)$ can be tabulated as (33) is being solved; the slope m_i and y-intercept c_i of node i are calculated using (36) after the optimal configuration is found for tree rooted at i, and they are then stored in dynamic programming table $M[\ ]$ and $C[\ ]$, similar to table $F_\lambda[\ ]$ used in solving (33). When the constraint variable is sufficiently close to the memory size, multiplier value $g_\lambda(r)$ is used instead.

5 Implementation & Results

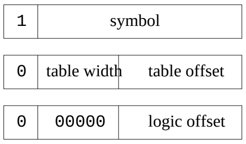
5.1 Implementation of Optimizer-Code Generator Pair

An optimizer serves as the front-end of our optimizer-generator pair. A VLC table containing the symbols and associated codewords and probabilities are input into the optimizer, along with the values of the parameters of the machine model that models the underlying processor. The optimizer parses the table and transforms it into a binary tree. It then performs the optimization described in Section 4. The computed configuration is passed on to the code generator.

In general, the computed configuration has a mixture of lookup tables and logical comparisons, and it is the code generator's job to implement the configuration using a mixture of C and native assembly code. Code generation for programmed logic is relatively straight-forward; a sequence of nested `if` statements with labels are generated corresponding to the section of the binary tree that uses logical comparisons. Code generation for lookup tables is more complicated, as tables need to reside in hierarchical memories corresponding to their memory assignments. In architecture where explicit cache movement is possible via native assembly codes (e.g. DEC Alpha), we can create lookup tables and assign them explicitly to the hierarchical memories as prescribed in the computed configuration. In other architectures such as the Pentium, we use the following approximation scheme instead, which creates lookup tables assigned to type 1 memory in a way that they will be more likely to reside in the L1 cache.

We first define an array p, large enough to contain all the elements in all the tables. For all the tables that are assigned to type 1 memory, we map the tables onto the array in breadth-first order, starting at the root of the tree. This will ensure that all type 1 memory assigned tables are in contiguous memory, and that each type 1 memory assigned mother-child table pairs are closer together than other type 1 assigned tables. We then do the same procedure for the type 2 memory assigned tables starting at the root of the tree. This ensures that type 1 memory assigned tables are more likely to be in the cache than type 2 memory ones.

The encoding scheme for each array element is shown in Figure 8a. If the most significant bit (MSB) of the element is 1, then we have reached the leaf of the tree, and the next 31 bits contain the symbol number. If MSB is zero and the next 5 bits are non-zero, then the 5 bits encode the width of the next table. The last 26 bits contains the offset in memory location of the next table relative to the first element of the array p. If the 5 bits are zeroes, then the last 26 bits contains the offset in memory location of the next programmed logic instruction relative to the first logic



a) Array element layout

```
#define initDec()
int off[1];
_asm lea EAX, logic0
_asm lea EBX, logic0
_asm sub EBX, EAX
_asm mov off[0], EBX
p[0]= -2147483648;
p[1]= -2147483647;
p[2]= -2147483646;
p[0+3]= off[0];
```

b) Initialization macro

```
#define match(index, A)
...  // table lookup
logic0:  if (A >= 0) {
index = 3;
_asm jmp done }
else {
index = 4;
_asm jmp done }
done:  ;
```

c) Decoding macro

Figure 8: Array Element Layout, Example of Generated Code

instruction.

The code generator first generates an initialization macro `initDec()`, written in C and in-line Pentium Assembly code, that performs initializations of all such array elements. It then generates the decoding macro `match(index, A)`, which performs the decoding procedure outlined by the configuration. In Figure 8, we see the example initialization macro and decoding macro for the configuration in Figure 2b. `match(index, A)` first performs a table lookup. If it is successful, then it will jump to `done` and return the correct symbol number in variable `index`. Otherwise, it will jump to `logic0` to perform a logical comparison.

5.2 Results

To test our algorithm, we ran our algorithm with two different sets of inputs. The first set of VLCs is the motion vector VLC table from H.263 video compression standard [9]. The longest codeword in this case is 13 bits. We fed the codebook and codeword probabilities into our optimizer-generator pair to generate an optimal VLC decoder. For the parameters of the cost model, we let $S_1 = 16kB$, $T_1 = 1$, $T_2 = 3$, and $Q = 0.5$, which are estimates for our testbed, a 266MHz Pentium processor.

To compare our approximate solution to the optimal, we use the pseudo-polynomial algorithm discussed in [12] to find the optimal solution. We first note that the execution of the approximate algorithm takes seconds on this data set, while the pseudo-polynomial algorithm takes 10-15 minutes. Note also the the pseudo-polynomial algorithm would be completely impractical for larger codebooks. When the optimal solution is found, we notice that the optimal configuration is the same as our approximate configuration.

For a test bit stream, we generated 10 million random codewords using the available codeword probabilities. Using our testbed machine, we execute our VLC decoder 20 times on the bit stream to obtain an average lookup speed. In Figure 9, we first compare the performance of our decoder to two simple decoders: single table lookup implementation, and logic only implementation discussed in

	Logic only	Optimal	Full Table	Moffat & Turpin
H.263 VLC	3.87	4.76	4.32	3.82
PTSVQ	2.67	4.16	3.70	-

Figure 9: Results for decoding TSVQ and H.263 VLC, in mil lookups per sec

the Introduction. we see our decoder is a 10.2% faster than the single table lookup implementation, and is 23.0% faster than the logic only implementation. Since the set of VLCs is a canonical code, we are able to compare our algorithm to algorithm ONE-SHIFT in [7]. We see in Figure 9 that our algorithm has a 24.6% improvement over algorithm ONE-SHIFT.

The second set of VLCs is the codebook of a pruned tree-structured vector quantizer. We obtained the TSVQ source code from [13]. The training set used for the construction of the codebook consists of three 512x512 grey-scale images: the well-known `lena`, `tiffany` and `baboon` images. Using the training set as input to the program, we obtained the PTSVQ codebook and the codeword probabilities for vector dimension 4 and rate 5. The longest codeword was length 15. In Figure 9, we see that we have 12.4% improvement over the full table lookup implementation and 55% improvement over the logic only implementation.

6 Conclusion and Future Work

In this paper, we described an optimizer-generator pair that generates a software VLC decoder for general purpose processors with hierarchical memories. We first formulated the problem as an optimization problem with respect to a machine model, then we showed the problem is NP-complete. We then presented a Lagrangian-based approximate algorithm with fast execution time. We showed that the performance of the generated implementation using a mixture of programmed logic and table lookups is superior to single table lookup implementation and logic only implementation. A possible future work is to address the problem of constructing a tree-structured vector quantizer such that it simultaneously minimizing the decoder's execution time as well as distortion of the decoded output.

References

[1] A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *PROC. IRE*, 40 (1952) 1098-1101.

[2] A. Gersho, R.M. Gray, *Vector Quantization and Signal Compression*, 1992.

[3] D. Hirschberg, D. Lelewer, "Efficient Decoding of Prefix Codes," *Communications of the ACM* vol.33, No.4, pp.449-59, April 1990.

[4] Kuo-Liang Chung, Yih-Kai Lin, "A Novel Memory-efficient Huffman Decoding Algorithm and its Applications," *Signal Processing*, Oct. 1997, vol.62, (No.2):207-13.

[5] Andrzej Sieminski, "Fast Decoding of the Huffman Codes," *Information Processing Letters*, No.26, pp.237-241, 1998.

[6] R.Hashemian, "Memory Efficient and High-Speed Search Huffman Coding," *IEEE Trans. Commun.*, vol.43, No.10, pp.2576-2581, October 1995.

[7] A.Moffat and A.Turpin, "On the Implementation of Minimum Redundancy Prefix Codes," *IEEE Trans. Comm.*, vol.45, No.10, pp.1200-1207, October 1997.

[8] R.Yeung, "Alphabetic Codes Revisited," *IEEE Trans. on Info. Theory* vol.37, No.3, pp.564-572, May 1991.

[9] Draft ITU-T Recommendation H.263, Video Coding for Low Bitrate Communication, 1995.

[10] Garey and Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, pp.224, 1979.

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

[12] G.Cheung and S.McCanne, "Optimal Routing Table Design for IP Address Lookups Under Memory Constraints," accepted to *Infocom 99*.

[13] C code of Tree-structured Vector Quantizer is publicly available at University of Washington Data Compression Laboratory, `ftp://isdl.ee.washington.edu/pub/VQ/code/`