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

Gene Cheung, Steve McCanne, Christos Papadimitriou
Department of EECS
University of California, Berkeley

Abstract

Implementation of variable-length code (VLC) decoders can involve a tradeoff between number of decoding steps 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. 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.

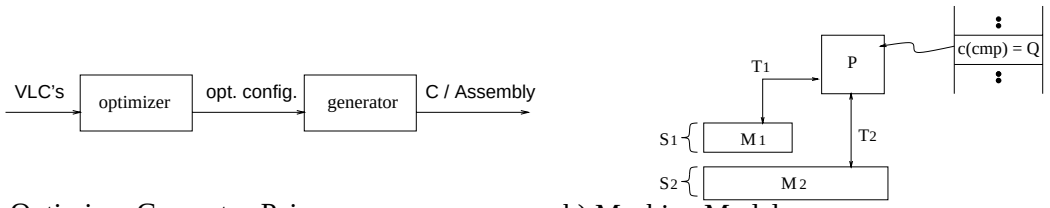
I. INTRODUCTION

The cornerstone of a wide variety of data compression algorithms is variable-length coding (VLC). Huffman code [1], for instance, maps a sequence of recurring, statistically independent symbols into a minimally described bit sequence by representing frequently occurring symbols with short codewords and rare symbols with longer codewords. Likewise, a pruned tree-structured vector quantizer (PTSVQ) maps an input vector into one of a finite number of codewords in a multi-stage approximation that produces short codewords for coarse vectors and longer codewords for finer-grained vectors. In either case, an *encoder* maps an information source into discrete codewords that are transmitted to a *decoder*, which in turn maps the codewords back to a discrete set of symbols that, perhaps only approximately, reconstitutes the original information source.

In many applications, the decoder's performance is critical and should thus be optimized. Unfortunately, the most straightforward and well-known method for decoding a set of prefix VLCs is quite suboptimal. In this approach, the set of codewords is represented as a binary tree, where each edge defines an input decision and a leaf node represents a terminal symbol. The decoder parses the next symbol from the input bit stream by traversing the tree from the root, following the corresponding edge for each bit in the input until a leaf is reached, i.e., the symbol is decoded. While this approach is space-efficient, as it only requires memory size proportional to the number of symbols in the codebook, it is time-inefficient because the cost of decoding each symbol is $O(d)$, where d is the the length of the longest codeword.

A more efficient approach can be realized by trading space for decoding steps. Rather than processing only one bit at a time, the decoder could process several bits in parallel using table lookups. In this approach, the decoder forms an index from the next d bits of input and use this index to locate an entry in a lookup table. The entry contains both the corresponding symbol and the length of the codeword identified. Accordingly, the decoder consumes the proper number of bits from the input and simultaneously determines the next symbol in the coded bit stream. While this approach is time-efficient as it requires an $O(1)$ processing cost to decode each symbol, its space requirements can be prohibitive as the lookup table requires $O(2^d)$ space.

Clearly, an "optimal" implementation would lie somewhere in between these two extremes and, in particular, would depend on how optimal performance is defined. In a hardware implementation, space might be more important, while in a software environment, memory might be cheap while



a) Optimizer-Generator Pair b) Machine Model

Fig. 1. Block Diagram of Proposed System and Machine Model

processing cycles scarce. Depending on these criteria, a number of previous works have proposed various schemes for exploiting this decoding steps / space tradeoff.

To minimize memory required to represent the Huffman tree, [3] [4] present compact data structures used 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.

We build on these previous approaches with a novel framework that systematically optimizes a VLC implementation to explicitly account for the tradeoff between decoding steps and space that is tailored for general purpose machine architectures (e.g., a Pentium PC). In our approach, a VLC decoder implementation is synthesized and optimized by carefully balancing the performance tradeoffs of memory accesses against iteratively programmed logic. Rather than restricting our implementation to a single lookup table or to a single tree-based data structure, our system decomposes the decoding algorithm into a mixture of multiple table lookup stages and imperative logic in a way that best matches the resource constraints of the target computing environment. For example, if we know parameters like the size of the CPU's on-chip processor cache, the relative cost of a cache miss (i.e. off-chip memory access), the cost of an imperative logical comparison, we could lay out a set of lookup tables that are congruent with the target environment's memory hierarchy. That is, given a machine model and a set of VLCs, we can synthesize an implementation that minimizes the expected time to decode a symbol by creating sub-tables and assigning these sub-tables to memories such that frequently accessed tables reside in local, fast cache memory, while larger, less frequently accessed tables reside in slower, larger memories. Note that we are solving the general case of decoding VLCs, a superset that includes the set of minimum redundancy codes. While minimum redundancy coding 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.

A typical computing configuration might consist of a very fast 16KB on-chip level 1 (L1) cache, a 512KB off-chip medium-speed level 2 (L2) cache, and a large 128MB DRAM slow-speed memory. Figure 1b, for example, illustrates how we might capture these characteristics with a parameterized machine model. Given this model, we can use the *optimizer-generator* pair of Figure 1a to optimally layout the decoding tables with respect to the machine model (the *optimizer*) and synthesize a native implementation for the target architecture (the *generator*).

To formally address this problem, we formulated a precise definition of the optimization problem in Section II. Unfortunately, this optimization problem turns out to be NP-complete, shown

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

Fig. 2. Example of a Configuration

in Section III). To avoid the combinatorial cost of the NP-complete optimal solution, we developed an approximate algorithm based on a particular Lagrangian-based optimization technique in [11]. Our approximate algorithm, presented in Section IV, has fast execution time and generates a VLC decoder whose performance deviates from optimality by a bounded amount. In Section V, implementation of the code generator is discussed, and results are presented.

II. Machine Model

Modern general-purpose 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 type of memory referenced. A machine model that reflect 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:

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

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 comparisons 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.

We can pose a well-formed optimization problem that we call the "VLC decode problem":

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

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.

	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

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

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 IV.

We will next show that even in the case when we use only lookup tables, the problem is NP-complete. So 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 IV.

III. 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 sketch an outline of the proof of NP-completeness for this decision problem. The details of the proof can be found in [13].

A. 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.

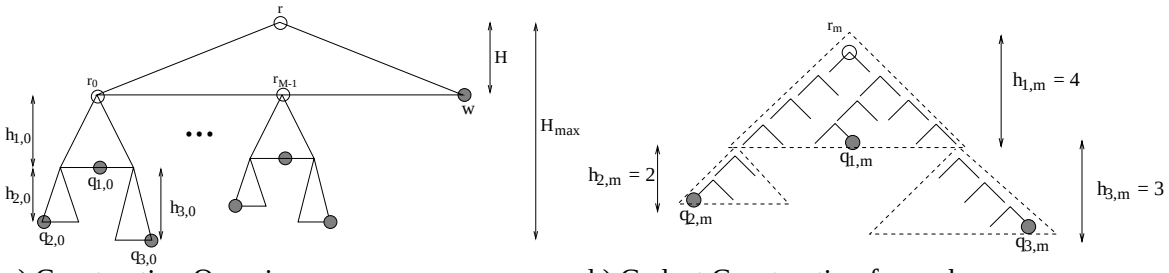
The same problem can be reformulated as the "partial sum" version as follows. Suppose we have M numbers in numeric base $M + 1$, each with $3N$ digits. We transform each 3-tuple (i_m, j_m, k_m) in the input list to a number $a_m = (M+1)^{2N+i_m} + (M+1)^{N+j_m} + (M+1)^{k_m}$, $\forall i_m, j_m, k_m \in \{0...N-1\}$. Notice each number has exactly three 1's in three digit positions and the rest of the digits are zeroes. Note 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.

B. Overview of Proof

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

Theorem 1: The VLC decode problem using only lookup tables and under a 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



a) Construction Overview b) Gadget Construction for each a_m

Fig. 4. Proof Constructs used in the NP-complete Proof of VLC decode Problem

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, $m \in \{0...M-1\}$. 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 mini-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 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, type 1 memory usage for 3-triangle configured subtree m is also a_m. We call the subtree configuration that uses one type 2 memory assigned lookup table for the entire subtree m the *default configuration*. 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. See [13] for the details of the NP-completeness proof.

IV. 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.

A. 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:

$$\min_{b \in B} H(b) + \lambda R(b) \tag{4}$$

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).

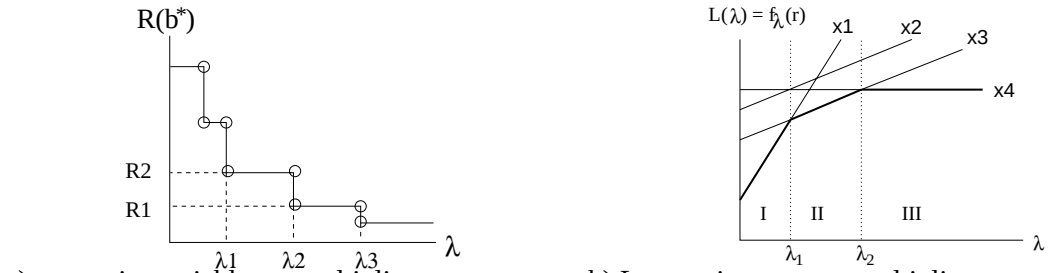
To solve (4) for a particular value of λ, we first represent the set of VLCs in question by a binary tree, where nodes are numbered in post-order with root r. We define a function $f_\lambda(i)$, which returns the minimum Lagrangian cost, $H(b) + \lambda R(b)$, of all possible configurations for the 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:

$$f_\lambda(i) = \min \left\{ w_i Q + \sum_{j \in L_{1,i}} f_\lambda(j), \min_{1 \le h \le H_i} \left[w_i T_1 + \lambda 2^h + \sum_{j \in L_{h,i}} f_\lambda(j) \right], \min_{1 \le h \le H_i} \left[w_i T_2 + \sum_{j \in L_{h,i}} f_\lambda(j) \right] \right\} \tag{5}$$

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 (5) 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 (4), 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 (5):

$$f_\lambda(i) = \min \left\{ w_i Q + \sum_{j \in L_{1,i}} f_\lambda(j), \min_{1 \le h \le \lfloor \log_2 S_1 \rfloor} \left[w_i T_1 + \lambda 2^h + \sum_{j \in L_{h,i}} f_\lambda(j) \right], w_i T_2 \right\} \tag{6}$$

We note that there are overlapping sub-problems when solving $f_\lambda(r)$ using (6); 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



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

Fig. 5. Constraint variable and Lagrangian cost as functions of multiplier

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 (6) and stores it in the table. After solving the Lagrangian problem using (6), 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 (4) 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:

$$|H(b_1) - H(b^*)| \leq |H(b_1) - H(b_2)| \tag{7}$$

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

B. 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 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 5a, 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 (5) that by construction, the optimal

configuration has Lagrangian cost of form:

$$f_\lambda(i) = \sum_{x \in X} w_x T_1 + \sum_{x \in X} 2^{h_x} \lambda + \sum_{y \in Y} w_y T_2 + \sum_{z \in Z} w_z Q \tag{8}$$

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:

$$f_\lambda(i) = c_i + m_i \lambda \tag{9}$$

$$c_i = \sum_{x \in X} w_x T_1 + \sum_{y \in Y} w_y T_2 + \sum_{z \in Z} w_z Q \tag{10}$$

$$m_i = \sum_{x \in X} 2^{h_x} \tag{11}$$

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 5b, 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. $temp := \min_{1 \le h \le \lfloor \log S_1 \rfloor} \left\{ I\left([c_i, m_i],\ [w_i T_1 + \sum_{j \in L_{h,i}} c_j,\ 2^h + \sum_{j \in L_{h,i}} m_j]\right) \right\}$
if $temp > \lambda$ **&** $temp < g_\lambda(i)$, **then** $g_\lambda(i) := temp$ `// check config. w/ type 1 memory table at node` i
3. $temp := I\left([c_i, m_i],\ [w_i T_2, 0]\right)$
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`

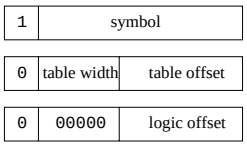
where function $I([c_1, m_1], [c_2, m_2])$ takes in the slopes m's and y-intercepts c's of two lines, and returns the intersection point. If they are parallel lines, it returns ∞.

$g_\lambda(i)$ can be tabulated as (6) is being solved; the slope m_i and y-intercept c_i of node i are calculated using (9) 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 (6). When the constraint variable is sufficiently close to the memory size, multiplier value $g_\lambda(r)$ is used instead.

V. Implementation & Results

A. 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



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

Fig. 6. Array Element Layout, Example of Generated Code

parses the table and transforms it into a binary tree. It then performs the optimization described in Section IV. 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 6a. 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 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 6, 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.

B. 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

	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	-

Fig. 7. Results for decoding TSVQ and H.263 VLC, in mil lookups per sec

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 7, we first compare the performance of our decoder to two simple decoders: single table lookup implementation, and logic only implementation discussed in 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 7 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 [14]. 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 7, we see that we have 12.4% improvement over the full table lookup implementation and 55% improvement over the logic only implementation.

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] G.Cheung, S.McCanne, C.Papadimitriou, same title to be submitted as Technical Report, Department of EECS, University of California, Berkeley, 1999.

[14] `ftp://isdl.ee.washington.edu/pub/VQ/code/`