AN EFFICIENT MEMORY CONSTRUCTION SCHEME FOR AN ARBITRARY SIDE GROWING HUFFMAN TABLE

Sung-Wen Wang, Shang-Chih Chuang, Chih-Chieh Hsiao, Yi-Shin Tung and Ja-ling Wu

CMLab, Dept. CSIE NTU, Setabox Corporation, Graduate Institute of Networking and Multimedia NTU
E-mail:{song,peiz,chchhsiao,tung,wjl}@cmlab.csie.ntu.edu.tw

ABSTRACT

By grouping the common prefix of a Huffman tree, in stead of the commonly used single-side rowing Huffman tree (SGH-tree), we construct a memory efficient Huffman table on the basis of an arbitrary-side growing Huffman tree (AGH-tree) to speed up the Hufman decoding. Simulation results show that, in Huffman decoding, an AGH-tree based Huffman table is 2.35 times faster that of the Hashemian's method (an SGH-tree based one) and needs only one-fifth the corresponding memory size. In summary, a novel Huffman table construction scheme is proposed in this paper which provides better performance than existing construction schemes in both decoding speed and memory usage.

1. INTRODUCTION

Traditional entropy coding in data compression applications widely relies on Huffman codes. Thus, most software approaches of entropy decoding are limited to sequential decoding because the representation of each symbol is variable in size. References [7]-[8] show that entropy decoding occupying a major portion of decoder's timing profiles. To speed up entropy decoding, however, is not as easy as to speed up the other modules of a decoding process by issuing several independent instructions simultaneously. Hashemian [1] proposed a method to construct memory efficient look-up-table (LUT) by clustering several bits together to speed up bit stream decoding. However, the binary tree associated with a Huffman code (i.e. the Huffman tree) grows sparser and sparser from the root because of the geometrical distributions of various media sources. This sparsity either causes the problem of memory waste or enlarges the latency of locating a leaf symbol. For a high sparsity Huffman tree, such as SGH-tree, Hashemian [2] proposed a condensed Huffman table (CHT) that represents the original Huffman table losslessly and compactly. In addition to Hashemian's methods, Jiang et al. [4] exploited the idea of [3] to partition an SGH-tree on the basis of pattern-matching and constructed a memory-efficient LUT. Specifically, the growth of SGH-tree is right- or left-side oriented, and it has regular leading bit-patterns 0_x1 (or 1_x0). The bit-pattern 0_x1 (or 1_x0) means there are x consecutive 0's (or 1)'s and ending by a 0 (or 1). By counting the number of leading 0's or 1's, they partition an SGH-tree into numbers of sub-trees. For each sub-tree, a small LUT is then constructed. The above-mentioned techniques perform the SGH-tree partitioning well. Nevertheless, most of nowadays video codecs elaborating Huffman tables to estimate each symbol precisely for dealing with realistic cases. As a result, the corresponding Huffman trees are often arbitrary-side growing and are more complicated than previously addressed ones. Thus, the approaches which are designed on the basis of SGH-tree may not work well now. To take both memory-efficient LUT and achieve high-speed locating of each symbol into account, we propose a new partitioning method for arbitrary-side growing Huffman table in Section 2. Section 3 demonstrates our simulation results. Finally, Section 4 concludes this write up.

2. ARBITRARY-SIDE GROWING HUFFMAN TABLE (ASHT)

We introduce two operations to partition a Huffman tree: The Hashemian cut (HC) and the bits-pattern-xor (BPx). These two operations are motivated by the Hashemian's method [1] and the bits-pattern matching scheme [4], respectively. HC is an operator that clusters the common length symbols together (c.f.: Fig.1 (b)). The HC behaves very similar to the scheme given [1] but the length of clusters is various. BPx is an operator that indexes each symbol by the number of leading common bits with a certain bits-pattern (c.f.: Fig.1(c)). The scheme given in [4] simply counts the leading 1's of symbols, so it is inefficient for an AGH-tree; therefore, a better memory usage can be expected.

More specifically, as shown in Fig.1(a), we take a Huffman tree [5][1], T_1, as an example to illustrate the operations of HC and BPx. HC partitions a tree by cut-lines. As shown in Fig.1(b), T_1 is partitioned by 3 cut-lines: x-x, y-y, and z-z.

This work was partially supported by the National Science Council and the Ministry of Education of ROC under the contract No. NSC 94-2752-E-002-006-PAE, NSC 94-2622-E-002-024, NSC 94-2213-E-002-078.

[1]This Huffman tree associates with a Huffman table is used in Windows Media Video 9, which estimates the probability of occurrence of different transform types at high bit-rate.

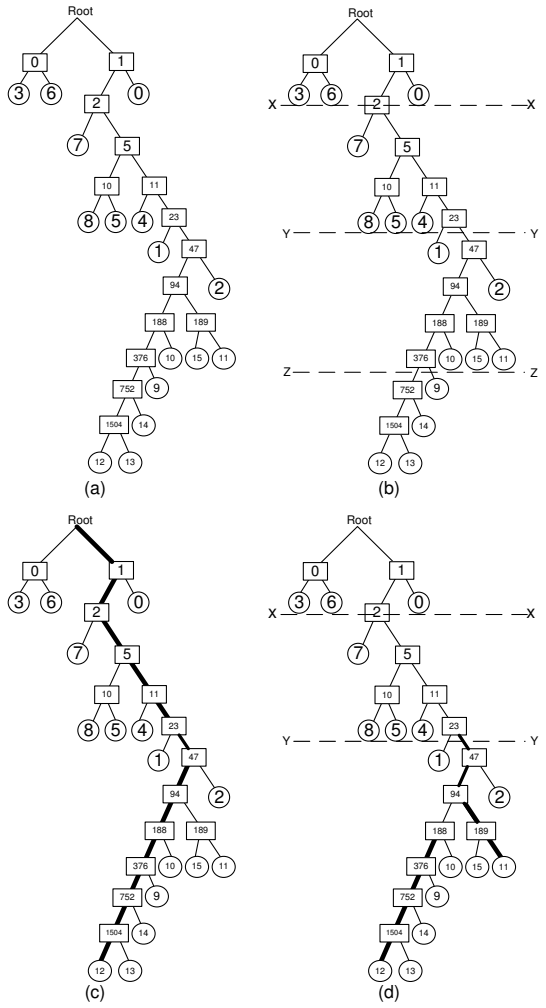


Fig. 1. The rectangular boxes represent the codeword values and the circle nodes represent the values of symbols. (a)A Huffman tree with 16 symbols, (b)An example of applying the HC method, (c)An example of applying the BPx method in which the boldface line represents one of the bits-patterns and (d)An example of ASHT clustering.

There is a LUT for each cluster which memorizes every value of symbols and every code length of symbols within the cluster. HC introduces memory waste because it duplicates the symbols that are not of the same length as the cluster size. For example, in Fig.1(b), symbol 7 in Fig. 1(b) will be duplicated four times in the second cluster. For a BPx, we index symbols by counting the largest number of common leading bits of a given bits-pattern. For $(1001)_b$ and $(1011)_b$, the largest number of common bits is two. In practice, the above operation can be realized by counting the number of leading zeros after bitwisely XORing with a given bits-pattern. Consider the example in Fig. 1(a), given the bits-pattern "$(101111000000)_b$" (as shown in boldface line in Fig. 1(c)), if we cluster all symbols of T_1 by BPx, the number of zeros of $\{3, 6\}$, 0, 7, $\{8, 5\}$, 4, 1, 2, $\{15, 11\}$, 10, 9, 14, and $\{12, 13\}$ are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 and 11, respectively. Likewise, we need small LUTs to distinguish the symbols that are of the same number of leading zeros, such as $\{3, 6\}$, $\{8, 5\}$, $\{15, 11\}$, and $\{12, 13\}$.

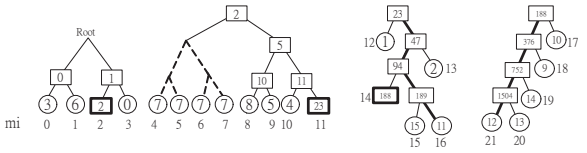
Due to the hardware consideration, the instruction for counting the leading zeros and bitwise-XORing are limited in length, we can not apply any length BPx onto a Huffman tree haphazardly. Meanwhile, if the symbols are of equal probability, the code lengths of them are the same after Huffman code construction. HC will perform better than BPx do, in this case. In the next section, we consider the two operations, HC and BPx, together to provide an even better Huffman tree partitioning.

2.1. ASHT Construction

Assume the length limit of instructions for counting leading zeros is 4 bits. For each root of a sub-tree one of the two prescribed operations can be applied. Fig. 1(d) gives an example in which an ASHT is constructed by combining HC and BPx together.

Intuitively, *T* is first cut by an HC with length-2, because there is a length-2 full sub-tree at the top of the Huffman tree. The remainder of *T* beyond the cut-line, x-x, forms another sub-tree rooted at codeword 2. For simplicity, we hereafter name the sub-tree rooted at codeword *cw* by *sb*(*cw*). As for *sb*(2), however, there is no apparent partitioning method. Because, if *sb*(2) is again partitioned by an HC with longer length or shorter length, the memory waste problem or the memory access time increasing problem will be introduced, respectively. If *sb*(2) is partitioned by a BPx with bits-pattern $(1111)_b$, the memory waste problem is solved but there still needs an extra memory access for *sb*(10). Here we apply an HC with length 3 to *sb*(2) because we are willing to utilize extra memory size for decreasing memory access. Similarly, *sb*(23) and *sb*(188) are respectively partitioned by bits-patterns $(1011)_b$ and $(0000)_b$, because there is no memory waste problem and needs no extra memory access.

To put ASHT into practical usage, the memory space as-



mi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
val	3	6		0	7	7	7	7	8	5	4		1	2		15	11	10	9	14	13	12
cl	2	2		2	1	1	1	1	3	3	3		1	2		4	4	1	2	3	4	4
sign	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
len			3																			
pat												11			0							
bl												4			4							
os			2									1			3							

(a)

mi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
val/pat	3	6	Nil	0	7	7	7	7	8	5	4	11	1	2	0	15	11	10	9	14	13	12
cl/len	2	2	3	2	1	1	1	1	3	3	3	4	1	2	4	4	4	1	2	3	4	4
sign	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
os			2									1			3							

(b)

Fig. 2. (a)The memory space associates with a given ASHT, and (b)The condensed memory space associates with (a). Where 'mi' denoted memory index while val, cl, sign, len, pat, bl, and os are defined in Section 2.1

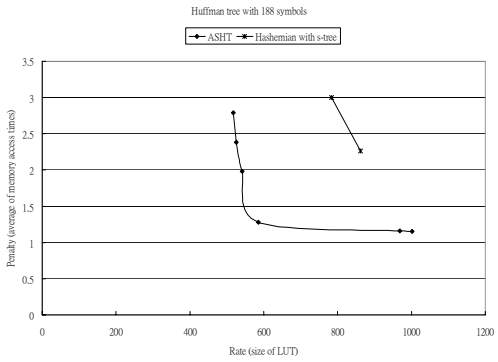


Fig. 3. For a given Huffman tree with 188 symbols, if the rate is set in the range of 800 to 875 Bytes, ASHT based approach can lessen the average memory access time from 2.35 to 1.90. On the other hand, if we fix the penalty ranging from 2.8 to 2.25, ASHT based approach decreases the required LUT size from (875 ± 37.5) to (520 ± 10) bytes.

sociated with ASHT needs to be considered. As shown in Fig. 1(d), if the symbol appears in the current cluster, ASHT needs only to memorize the symbol value and the symbol codelength. We use *val*-field to represent the value of a symbol and *cl*-field the codelength of a symbol. For the symbols that are not in the current cluster, we have to record the kinds of partitioning method that are applied. We use a sign bit 1 to indicate that the symbol is not in the current cluster. Meanwhile, it is necessary to record the length of HC or the bits-pattern of BPx. For an HC, we denote its length by *len*-field. As for BPx, we use *pat*-field to represent bits-pattern and *bl*-field the corresponding codelength. We also use the offset-field, denoted as '0', to represent the offset of memory address between the current cluster and the next one. An example of the memory space associated with an ASHT is shown in Fig.2(a).

For the purpose of condensing LUT, as shown in Fig. 2(b), we combine those fields together which do not function simultaneously. The functionalities of *len*, and *bl* are similar and they do not activated simultaneously, so we combine these two fields together. Similarly, we can combine *val* and *pat* together and use a sign bit to distinguish between them. To remove the ambiguity between a BPx with length *j* leading zeros and an HC with length *j*, we set *val* as Nil or not to distinguish them.

3. EXPERIMENTAL RESULTS

In this section, we compare the performance (in terms of execution speed and memory usage) of the proposed ASHT with that of the Hashemian's work [1]. Four tested Huffman trees are taken from [5], and they respectively have 188, 135, 73 and 16 symbols. The small set Huffman tree is exactly the one shown in Fig. 1(a). The curves shown in Figs. 3, 4, 5 and 6 are the convex hulls drown on the basis of the randomly generated ASHTs respectively for the four tested Huffman trees. The corresponding Hashemian curves are generated similarly. Notice that the penalty function (i.e. the vertical axis) and the rate (i.e. the horizontal axis) depicted in the above figures are the average time of memory access (which is proportional to the execution speed) and the LUT size(which is proportional to the memory usage), respectively. From the figures, the ASHT-based approach performs much better, both in terms of execution speed and memory usage, than that of the conventional (i.e., Hashemian's) one.

4. CONCLUSION

As shown in previous sections, the AGH-tree prevails over the SGH-tree in variable length codec design, which makes most of nowadays SGH-tree based code table construction algorithms improvable. Therefore, we investigate a new code table construction scheme, called ASHT, which provides better memory usage and needs less memory access. ASHT construction consists of two simple functions: HC and BPx. The characteristics of HC make it suitable for partitioning equal probability symbols; on the other hand, BPx performs better for symbols with geometrical distributions. By properly combining HC and BPx, ASHT results in the most efficient memory access and usage in the literature. However, we noticed that the number of possible ASHT structures are quite large for a Huffman tree with a large number of symbols. Thus, an optimization scheme for finding the best ASHT structure is

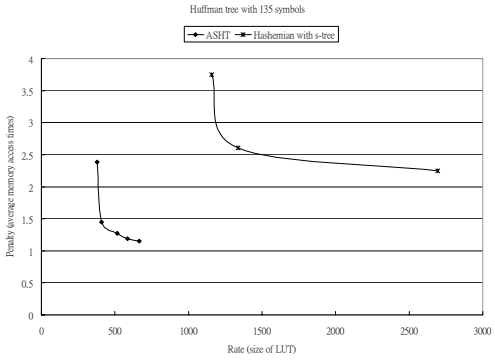


Fig. 4. For a given Huffman tree with 135 symbols. If we fix the penalty to be 2.4, the ASHT based approach decreases the LUT size 5 times (i.e., from 1900 down to 380 bytes).

very critical for designing the best code table and this is, of course, one of our future research directions.

5. REFERENCES

[1] R. Hashemian, "Memory Efficient and High-speed Search Huffman Coding," *IEEE Trans. on Comm.*, Vol. 43, No. 10, Oct. 1995, pp. 2576-2581.

[2] R. Hashemian, "Condensed Table of Huffman Coding, a New Approach to Efficient Decoding," *IEEE Trans. on Comm.*, Vol. 52, No. 1, Jan. 2004, pp. 6-8.

[3] S. B. Choi et al., "High-Speed Pattern Matching for a Fast Huffman Decoder," *IEEE Trans. on Consume Electronics*, Vol.41, No. 1, Feb. 1995, pp. 97-103.

[4] J.H. Jiang et al., "An Efficient Huffman Decoding Method Based on Pattern Partition and Look-up Table, "*APCC/OECC'99*, Vol 2, Oct. 1999 pp.904-907

[5] SMPTE VC-1 Video coding expert group (VCEG), ITU

[6] Sridhar Srinivasan et al., "Windows Digital Media Division, Microsoft Corporation, Windows Media Video 9: overview and applications,"*Signal Processing: Image Comm.,* Oct 2004.

[7] X. Zhou, et al., "Implementation of H.264 Decoder on General-Purpose Processors with Media Instructions", *Proc. of SPIE Con. on IVCP,* vol. 5022, Jan. 2003.

[8] Wang, S.-W., et al., "The optimization of H.264/AVC baseline decoder on low-cost TriMedia DSP processor, "*Photonic Devices and Algorithms for Computing VI. Proc. of the SPIE*, Vol. 5558, pp. 524-535 (2004)

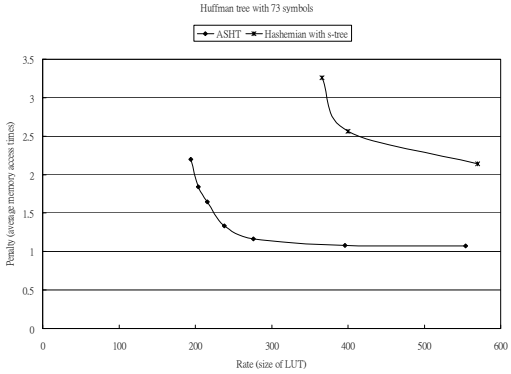


Fig. 5. For a given Huffman tree with 73 symbols. If the rate is set in the range of 336 to 556 bytes, ASHT based approach can lessen the average memory access time from 2.35 to 1.90. On the other hand, if we fix the penalty to be 2.19, the ASHT based approach decreases the required LUT size from 556 to 194 bytes.

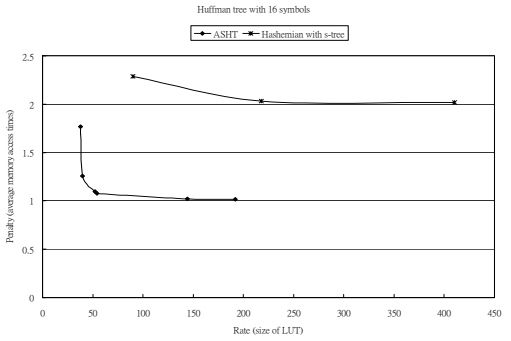


Fig. 6. For a given Huffman tree with 16 symbols, if the rate is set in the range of 90 to 192 bytes, ASHT based approach can lessen the average memory access time from 2.15 to 1.61.