

OPTIMIZED HUFFMAN DECODING FOR AAC

VLADIMIR Z. MESAROVIC , RAGHUNATH RAO, MIROSLAV V. DOKIC & SACHIN DEO

Cirrus Logic Inc , Austin, TX, USA

In today's competitive consumer audio market the Advanced Audio Coding (AAC) format has quickly become a must-have technology with its adoption on the Internet, in digital radio, digital television and home theatre. Compression using AAC retains high audio quality even at low bit rates. One reason for this effectiveness is the use of Huffman variable length coding to represent frequency domain information. However, this requires to perform the relatively complex task of Huffman decoding in the audio decoder, which is typically very sensitive to cost and processor speed requirements. Furthermore, encoders can sometimes create worst-case scenarios consisting of very long code words, even when unnecessary. Thus, one needs to optimize the Huffman decoding for these worst-case scenarios without giving up average performance. This paper discusses various methods for Huffman decoding, their inherent implementation tradeoffs on a DSP platform and proposes improvements that are specific to the Huffman codebooks used in AAC.

1. INTRODUCTION

In the last decade we have witnessed an unprecedented growth of the digital audio. Driven by clear economic factors for more efficient utilization of the existing bandwidth as well as mass storage, and enabled by powerful and cost-effective DSP and memory technologies, the engineering community has developed several audio compression standards that cover a whole range of applications. Major application areas include cinema and home theaters, digital broadcast, internet and portable applications, just to name a few.

The most comprehensive digital audio standard yet is the ISO/IEC 14496-3 [1], better known as the MPEG-4 Audio standard. Unlike its predecessors, MPEG-1 and MPEG-2 Audio coding standards, and unlike many speech coding standards, MPEG-4 Audio prescribes a complete set of tools for coding low and high bitrate, natural and synthetic speech and music. A major part of natural MPEG-4 Audio is general audio coding, which deals with medium to high quality audio. The general audio coding subpart of MPEG-4 [1] is based on MPEG-2 AAC [2] standard. MPEG-2 AAC coding has quickly become very popular in industrial circles due to its audio quality.

For example, MPEG-2 AAC has been adopted as the main audio coding standard for Japanese satellite and terrestrial HDTV systems. In the portable player market, the AAC algorithm is one of the top contenders for the technology of choice along with the MP3 algorithm. In the near future, the AAC technology is expected to find many other applications based on the versatility of the MPEG-4 international standard.

Recently, many semiconductor manufacturers rushed to provide the AAC audio decoder solution for these rapidly growing market segments. Cirrus Logic, Inc. with its Crystal Audio Division responded swiftly with the introduction of a 5.1-channel Low Complexity (LC) profile AAC solution with the CS49300 chip family [3]. This decoder is currently being used in Japan for both HDTV and digital audio broadcast applications as well as in the audio-video outboard decoder devices.

There are at least three reasons which make cost-effective AAC decoder implementations difficult. First, the AAC algorithm was designed to be implemented on a 32-bit floating-point engine. Second, the memory requirements (especially in the

multichannel case) are significantly large. And third, the computational requirements to support all profiles, tools, and bitrates are very high. Nevertheless, cost-efficient (24-bit fixed-point DSP) AAC implementations with relatively moderate memory resources are possible, but they have to properly balance the MIPS/memory trade-offs, while maintaining full data precision.

In [4], the algorithmic and numerical details of 32-bit floating-point calculations on a 24-bit fixed-point dual-DSP architecture were addressed. This architecture was shown to represent a significant cost reduction over 32-bit floating-point DSP AAC implementations, while preserving full data precision. In addition, an AAC algorithm split between the 2 DSPs was proposed in [4], and some of the subtask-implementations were discussed in details. More specifically, degrouping, inverse quantization, and scaling were optimized on the front-end DSP in the 24-bit fixed-point architecture. The back-end DSP was responsible for the inverse modified discrete cosine transform (IMDCT), windowing and PCM buffer management.

In [5] the problem of efficient AAC filterbank and PCM buffering/playback implementation using the same DSP architecture as in [4] was addressed. Two schemes for efficient PCM buffer management were proposed. The first approach minimized data memory requirements by utilizing a non-uniform size PCM buffering. These optimizations were based on AAC decoding dynamics and the available MIPS budget. The second approach overcame severe memory shortages by allowing integration of (additional) external memory to the AAC decoder while maintaining minimal complexity and MIPS impact. In this approach the MIPS overhead of transferring data to/from external memory were further minimized by using the on-chip Direct Memory Access (DMA) engine.

In this paper we discuss optimization of the Huffman decoding strategy for AAC decoders. We review and analyze standard Huffman decoding implementations in the AAC context and propose a new table lookup approach which significantly reduces the computational requirements. Finally, we propose the "optimal" approach (out of the 3 methods described) for each of the 12 AAC Huffman codebooks.

The rest of the paper is organized as follows. In Section 2 we review two traditional Huffman decoding approaches (the linear and the binary search) and propose a new two-step table lookup method. In Section 3 we compare those three methods in terms of memory and MIPS requirements and make proposals for selecting the "optimal" approach for all 12 Huffman codebooks. Finally, conclusions are given in Section 4.

2. REVIEW OF HUFFMAN DECODING APPROACHES

Unlike some other compression technologies (such as DTS, for example), the AAC algorithm has mandated the Huffman coding tool at the encoding side in order to further reduce the compressed data bitrate. This requirement forces the decoder to always perform Huffman decoding of the received bitstream. This is true, even in cases where application allows for high bitrates where compression gains are not high. In these cases encoders do not tend to be very parsimonious, i.e. non-efficient. By non-efficient encoding we mean here that encoders do not necessarily choose the "optimal" combination of scaling, quantization and Huffman codebook which result in the smallest amount of bits that satisfy prescribed audio quality. Rather, these non-optimized encoders use all available bits regardless whether perceivable quality is gained or not, and regardless whether the Huffman encoding further compresses or expands the data. For example, this situation may arise if the encoder is forced to utilize all available bitrate despite the fact that the encoded content is silence where additional bits are not further improving audio quality. As a consequence of this suboptimal choice the Huffman encoder is not necessarily reducing the number of bits with respect to the amount of information present in the input, thus resulting in bitrate expansion rather than compression. In other words, the Huffman coding gain (ratio of input and output bits in the Huffman encoder) may not be larger than 1. In these situations a decoder not only has to process lots of bits (high bitrate), but also has to perform "non-efficient" Huffman decoding, which may create additional MIPS requirement. By non-efficient Huffman decoding we mean here that longer words (that require more processing) will be decoded more frequently than if the Huffman coding gain was larger than 1.

The above situation suggests that robust decoders should have a very flexible Huffman decoding strategy, i.e., they have to be able to decode both low and high bitrate streams with either efficient or non-efficient Huffman encoding. One obvious solution to preclude this problem is to impose a constraint on the encoder to always produce efficient Huffman codewords, thus, simplifying decoder's procedure. Although this solution may be technically easiest to implement, practically speaking it may be the most difficult one to enforce. Namely, in the open international standards such as ISO/IEC 14496-3 [1] and ISO/IEC 13818-7 [2], there are no official certification procedures for either encoders or decoders. Therefore, AAC encoders are practically allowed to use any coding configurations as long as they produce syntactically legal bitstreams. AAC decoders, on the other hand, are only expected to pass a very limited number of compliance bitstreams to be called AAC compliant decoders. This in effect, creates an open

loop between the encoder and the decoder, which as a final result not only affects the encoder and decoder manufacturers but mostly the end users (customers). The best AAC decoder manufacturers try to distinguish themselves by supporting all possible encoding configurations, while trying to be cost-effective, i.e. not overdesigning the hardware and software resources.

Next we discuss Huffman decoder implementations that can alleviate problems with various encoder configurations. More specifically, we compare two of the commonly used Huffman decoding implementations (linear and binary search), with the new proposed method (two-step table lookup), and discuss their weaknesses and strengths in the context of AAC decoder implementation. The objective is to choose a Huffman decoding scheme that is optimized to minimize MIPS for worst-case scenarios without giving up average performance, with an acceptable trade-off of table storage.

2.1. Linear Search

This approach assumes that the decoder has stored three pieces of information for each codeword: the codeword itself, the length of the codeword (in bits), and the corresponding quantization index (output of the Huffman decoder). The codewords are stored in a table in their length non-decreasing order (shorter words first, followed by the longer ones), and their respective lengths (in bits) and output indexes are stored in separate tables. The decoder starts the decoding process by first reading the smallest possible number of bits from the bitstream (equal to the length of the shortest codeword in that particular codebook) and sequentially tries to match the extracted bits with the codewords of the same length stored in the codeword-table. If matching fails for all codewords of current length, the decoder grabs additional bits (according to the codeword-length table), concatenates them to the previously extracted bits and continues the search process "down" the codeword-table (sequentially considering longer code words) until the match is found. The quantization level corresponding to the matching codeword is read from the quantization index table and one Huffman codeword has been decoded. The process is repeated till the all the desired Huffman codes have been decoded.

In order to reduce the table storage requirements the codeword, its length field, and output quantization level (index) sometimes can be packed together in a single word. For example, the Huffman codebook 1 in [1, 2] has codewords at most 11-bits long, the maximum quantization level is 81 (takes 7 bits to represent), and the length field can be represented by 4 bits ($\text{length} \leq 11$ bits). Thus, for every table entry, we can combine $11 + 7 + 4 = 22$ bits together into, for example, the 24-bit data memory word. This packing allows for memory-efficient Huffman

codebook representation, while adding negligible DSP processing overhead. The table entries are typically arranged in the codeword-length non-decreasing order, so that the decoder checks for the most probable codewords first. Because of this, the length fields in the packed Huffman codebook tables can be further differentially encoded, so that only the additional number of bits to parse from the bitstream (with respect to the previous table entry) are stored.

Clearly, if a particular Huffman table is small, this approach is quite acceptable, since both memory and MIPS requirements are negligible, while the decoding approach is straightforward to implement. For larger Huffman tables, even though the table storage requirements can be substantially reduced by combining the three tables into one, the linear nature of the search may impose a significant MIPS overhead to the DSP processor, especially in the high-bitrate and non-efficient Huffman coding cases. In the worst case, the decoder would have to traverse through all table elements to find the match. Thus, this approach has linear complexity with respect to the codebook dimension.

2.2. Binary Search

In this approach the linear computational complexity from the linear search is practically reduced to a logarithmic order (actually proportional to the number of bits in the codeword), which for large Huffman tables represents significant improvement. This is achieved by creating a hierarchical binary tree table in which each subsequent node (child) is chosen depending whether a '0' or '1' is extracted from the bitstream. If the child is not a leaf (indicated by bit stored in the corresponding table entry), this process is continued by extracting successive bits. Once a leaf is reached, the decoder recovers the quantization index from the table.

This approach saves MIPS (with respect to the linear search) regardless of the efficiency of the Huffman encoder. Although in this approach the actual codewords are not stored in tables, this approach typically requires similar amount of memory for tables as the linear search, especially if we compare packed tables. The reason is that for every bit grabbed from the bitstream the decoder has to know at what node to continue and this branching pointer information is stored in the table. In addition, the information of whether or not this node is a leaf is also stored. If low memory resources and moderate MIPS resources are available this approach could be the method of choice.

2.3. Table-Lookup

In theory, the Huffman decoding could be done through a single table lookup by extracting the maximum number of bits required for a particular table and looking up the table for the quantization index. Clearly, for large Huffman tables (very long

codewords) this approach requires prohibitively large storage space. For example, in Huffman codebook 9 in [1, 2] the longest codeword is 15 bits, thus, a table with $2^{15} = 32768$ elements is needed to cover all possible cases. In most of the cases the decoder will be overdrawing bits which need to be reused for the subsequent decoding, which adds a very small MIPS overhead. Thus, a one-step table lookup approach represents the least MIPS-expensive solution for Huffman decoding, with prohibitively large memory requirements for large tables.

Extending the table lookup philosophy to minimize processing requirements, we propose an approach in which more (than one) bit grabs and more (than one) table lookups are used. For simplicity purposes, in this paper we will concentrate on the two-step table lookup. More specifically, we propose to design two tables per Huffman codebook, which will be used in the following fashion. The decoder would always extract certain number of bits, N , which is typically the length of the shortest codeword in the codebook. The first table would consist of 2^N elements which would be addressed by the first N extracted bits. The entry in this table would contain the base address of the second table and the number of additional bits, M , that decoder

needs to extract from the bitstream. The decoder then converts extracted bits to a numerical offset and adds it to the base address. The resulting address is the index to the second table from which the decoder obtains the quantization index and the original length of the corresponding codeword. The original (true) length of the matched codeword is to be compared to the number of extracted bits ($N+M$), so that overdrawn bits (if any) are restored and used for subsequent decoding. Note that with the special case of N =number of bits in the longest code word, the above scheme reduces to the single table lookup case.

Let us consider Huffman codebook 11 in [1, 2] as an example. This codebook has 289 codewords and its shortest and longest codewords are 4 and 12 bits long, respectively. So the binary search method will take at most 12 steps and requires a table of 289 elements. Clearly, the linear approach may be too MIPS consuming since in the worst case the decoder would have to traverse to the bottom of the table (289 elements) just to decode one quantization index. Again, if these least probable words are occurring very rarely, as it should be the case in the efficient Huffman coding case, the MIPS impact would be minimal. However, as practice has shown, not all AAC encoders are optimized and have produced bitstreams with very non-efficient usage of Huffman codewords. Since no certification procedures are enforced on the AAC encoding side, a safe approach from the decoder point of view is to use table-lookup approach for decoding Huffman codebook 11. If N is chosen to be equal 4,

the length of the first table is $2^4 = 16$ and the length of the second table is 476. This may be a price worth paying (in memory) for the "minimal" and always fixed processing per decoded word. More specifically, there are always two bit-field extractions from the stream and two table lookups, thus, with the exception of restoring potential bit overdraws, the decoder spends minimal (and equal) amount of instructions per codeword, regardless of the likelihood of appearance of the particular word and its length.

3. RESULTS

In this section we report our analysis of 12 Huffman codebooks used in AAC standards [1, 2]. Eleven Huffman codebooks (labeled HCB_1 - HCB_11 below) are used to encode subband samples, and one Huffman codebook (labeled HCB_SF below) is designed specifically for scale factors.

It is possible to characterize the exact number of instructions (MIPS) used for each Huffman decoding approach only in a statistical sense, i.e., through averaging results over ensemble of test vectors with different statistical properties. Here, we resort to qualitative reasoning to suggest an "optimal" decoding method for each codebook based on the algorithmic complexity.

We present memory requirements for the two-step table lookup approach for each Huffman codebook. For this approach we first find the minimum-size table by searching for the optimal number of bits to extract in the first table lookup, N . Then, assuming optimal N for the two-step table lookup method, and the most efficient word-packing for the linear and binary search approaches, we compare all three Huffman decoders in terms of overall MIPS and memory requirements, and propose the "best" approach.

3.1. MIPS Requirements

The MIPS requirements for the linear search grow linearly with the length of the received codewords. Therefore, we put the linear Huffman decoder into the HIGH MIPS category. It is easy to see that this Huffman decoder can have orders of magnitude higher computational requirements than the other two methods. For example, in case of the Huffman codebook 11, in the worst case, the decoder not only would have to extract all 12 bits one by one, but would have to compare all 289 words in the codebook to the received codeword.

The MIPS requirements for the binary search Huffman decoder are directly proportional to the number of extracted bits per codeword. Thus, the overall MIPS requirement for this approach falls under the MEDIUM MIPS category.

The two-step table lookup approach requires (almost) constant low MIPS budget for every extracted word. The instructions involved in this approach are mainly of table lookup in nature, except for the reuse of the overdrawn bits, when appropriate. The overall MIPS requirement for this approach clearly falls under the LOW MIPS category.

Although, the single table lookup scheme will have lowest MIPS, we do not consider it here since the storage requirement is prohibitive. Thus, from the MIPS perspective the two-step table lookup approach is the best among the three. Next, we look at the memory requirements for all three decoders and we propose the overall most acceptable solution.

3.2. Memory Requirements

The total memory requirements (for each Huffman codebook) for the two-step table lookup Huffman decoding approach are given in Tables 1 and 2. These memory requirements are computed for various values of N, where N denotes the length of the lookup index to a first table, and the "optimal" N (minimum memory) for each of the Huffman codebooks (HCBs) are chosen. Memory requirements (in words) only for optimal and close-to-optimal values of N are presented in Tables 1 and 2, others are left blank.

	N=3	N=4	N=5	N=6	N=7
HCB_1	-	184	160	192	-
HCB_2	121	113	121	164	-
HCB_3	-	4146	2136	1184	795
HCB_4	-	312	226	232	-
HCB_5	-	560	344	288	356
HCB_6	-	193	166	192	-
HCB_7	-	305	218	224	-
HCB_8	-	129	123	166	-
HCB_9	-	2112	1142	714	603
HCB_10	-	375	292	290	373
HCB_11	-	492	408	411	-
HCB_SF	-	-	-	-	4363

Table 1: Memory requirements (in words) for the two-step table lookup Huffman decode approach.

	N=8	N=9	N=10	Optimal N
HCB_1	-	-	-	5
HCB_2	-	-	-	4
HCB_3	793	1171	-	8
HCB_4	-	-	-	5
HCB_5	-	-	-	6
HCB_6	-	-	-	5
HCB_7	-	-	-	5
HCB_8	-	-	-	5
HCB_9	748	-	-	7
HCB_10	-	-	-	6
HCB_11	-	-	-	5
HCB_SF	2571	2060	2572	9

Table 2: Memory requirements (in words) for the two-step table lookup Huffman decode approach (continued).

Table 3 compares the memory requirements for the three approaches: (a) Linear search, (b) Binary search, and (c) Two-step table lookup. For comparison purposes minimal memory requirements (assuming most-efficient word packing) are used for all three methods.

	Linear search	Binary search	Two-step table lookup	"Optimal Method"
HCB_1	81	81	160	Binary
HCB_2	81	81	113	Binary
HCB_3	162	81	793	Binary
HCB_4	81	81	226	Binary
HCB_5	162	81	288	Binary
HCB_6	81	81	166	Binary
HCB_7	64	64	218	Binary
HCB_8	64	64	123	Binary
HCB_9	338	169	603	Binary
HCB_10	169	169	290	2-Step
HCB_11	578	289	408	2-Step
HCB_SF	242	121	2060	Binary

Table 3: Memory requirements comparison of Linear search, Binary search, and Two-step table lookup Huffman decoding approaches. The "optimal" method (most acceptable method) is chosen based on both MIPS and memory requirements.

4. RESULTS

Huffman decoding is a critical issue in the design of an AAC decoder since the Huffman coding tool is always active regardless of the bitrate setting. Non-efficient Huffman encoding can, in practice, end up in worst-case scenarios for Huffman decoding by consistently using very long codewords, sometimes even when unnecessary. To optimize the decoding for such worst-case scenarios without giving up average performance, for each Huffman codebook one can choose from the various decoding methods outlined above. In this paper we compared three Huffman decoding methods in terms of memory and computational requirements, and made suggestions for

their implementations for all 12 AAC Huffman codebooks in [1, 2]. The choice of implementation was based on a practical trade off between MIPS usage and memory requirements. In 10 of these 12 cases, binary search turned out to be optimal and in the remaining 2 cases, a two-step table lookup scheme proves to be most appropriate.

REFERENCES

- [1] ISO/IEC JTC1/SC29/WG11, "ISO/IEC FDIS 14496-3 Subparts 1, 2, 3, Coding of Audio-Visual Objects -Part 3: Audio", ISO/IEC JTC1/SC29/WG11 N2503, Oct. 1998.
- [2] ISO/IEC JTC1/SC29/WG11, "ISO/IEC IS 13818-7 (MPEG-2 Advanced Audio Coding, AAC)" ISO/IEC JTC1/SC29/WG11 N160, April 1997.
- [3] Cirrus Logic Crystal CS49300 data sheet, Dec. 1999, available from <http://www.cirrus.com>.
- [4] V. Z. Mesarovic, N. D. Hemkumar, M. V. Dokic, "MPEG-4 AAC Audio Decoding On a 24-Bit Fixed-Point Dual-DSP Architecture", *ISCAS-2000*, May 28 - Jun 1, 2000, Geneva, Switzerland.
- [5] V. Z. Mesarovic and M. V. Dokic, "Memory-Optimized AAC Decoder", *ISPACS-2000*, November 8-11, 2000, Honolulu, Hawaii.