Real time implementation of OGG VORBIS decoder on Analog Devices SHARC ADSP-21364

Sanjay Rajashekar, Ashish Shenoy P., Siddhartha Sampath, Akella Karthik and Dr. Sumam David S.
Department of Electronics & Communication Engineering
National Institute of Technology Karnataka, Surathkal, Mangalore 575025 INDIA
Email: sumam@ieee.org

***Abstract*—A real-time implementation of Ogg Vorbis audio decoder using Analog Devices SHARC ADSP-21364 processor is presented in this paper. The ADSP-21364 is a floating-point processor optimized for professional audio applications. We exploit the architectural features of this processor to implement the decoder using floating point arithmetic. Since the SHARC ADSP-21364 processor poses stringent memory constraints for the dynamic probability models used in Ogg Vorbis, a technique for significantly reducing memory requirement is presented. Compressed-array-representations of tree structures are used to reduce the execution time of the critical section of the code. This is one of the early implementations of floating point Ogg Vorbis decoder on the SHARC ADSP-21364 processor. Our implementation at 80 MIPS provides the basis for the development of an Ogg player capable of playing Ogg Vorbis audio files in real-time.**

I. Introduction

The Ogg Vorbis [1] [2] is a fully open, non-proprietary, patent-and-royalty-free, general-purpose compressed audio format for mid to high quality (8kHz-48.0kHz, 16+ bit, polyphonic) audio and music at fixed and variable bit rates, from 16 to 128 kbps/channel. This places Vorbis in the same competitive class as audio representations such as MPEG-4 (AAC), and similar to, but higher performance than MP3, TwinVQ, WMA and PAC. It has clear advantage over other lossy codecs in that it produces smaller files than most other codecs at equivalent or higher quality. Vorbis is the first of a planned family of Ogg multimedia coding formats being developed as part of *Xiph.org Foundation's* Ogg multimedia project. The latest official version is 1.2.0 released on 25 July 2007.

The term *Ogg* refers to a general purpose data container format. Ogg container format encapsulates *Vorbis*-encoded audio data. Hence the name *Ogg Vorbis*. Ogg uses octet vectors of raw, compressed data *(packets)*. These compressed packets do not have any high-level structure or boundary information; strung together, they appear to be streams of random bytes with no landmarks. But the structure in these data packets is implicit and is based on certain fields in the Ogg page header.

Vorbis I is a forward-adaptive monolithic transform codec based on the Modified Discrete Cosine Transform. The codec is structured to allow addition of a hybrid wavelet filterbank in Vorbis II to offer better transient response and reproduction using a transform better suited to localized time events. The Vorbis codec design assumes a complex, psycho acoustically-aware encoder and simple, low-complexity decoder. Vorbis decoding is computationally simpler than MP3, although it does require more working memory as Vorbis has a dynamic probability model [2]. The probability model is dynamic in the sense that a particular distribution is decided based on the audio content in that particular frame of data. There are many such models for an audio file, each frame being associated with a model. The larger the number of statistically different frames of data, larger will be the number of models used. Naturally, the information regarding these models have to be coded and sent along with the compressed audio data. Usage of dynamic probability model results in relatively better decoded audio quality, but requires larger working memory. This is because the coded information regarding these models has to be decoded during *codec setup* phase, stored in the heap, and referred to during *audio decode* phase. This makes the implementation of the Ogg Vorbis decoder on an embedded platform a challenging task.

Real-time decoding of Ogg Vorbis using fixed point arithmetic using specific hardware Line spectrum pair module along with ARM7TDMI processor [3], Texas Instruments TMS320C6416 using Altera Stratix II FPGA as hardware accelerator [4], Texas Instruments TMS320C5510 and Motorola 56002 have been tried using the Tremor fixed point reference code [5]. This work attempts to implement Ogg Vorbis decoder using floating point arithmetic using ADSP-21364 processor in a memory contrained environment.

Section II explains the principle of Ogg and Vorbis decoding. Section III discusses issues in migration from the available reference implementation to the real-time implementation on ADSP-21364, along with an optimised Huffmann tree representation. The experimental and resource utilisation results are presented in Section IV.

II. Ogg and Vorbis Decoding

Ogg and Vorbis are two separate entities. Vorbis is the codec which accepts PCM audio samples and outputs compressed audio packets. It also generates data packets for codec initialization. Ogg is a general purpose container format for any kind of data. It provides framing, synchronization, error detection and sync recovery. Decoding Vorbis embedded in Ogg is a three-step process as shown in Fig. 1. Synchronization and streaming layers deal with Ogg decoding. The output of the streaming layer is a reconstructed Vorbis packet which is fed to the Vorbis decode engine to obtain PCM audio data.

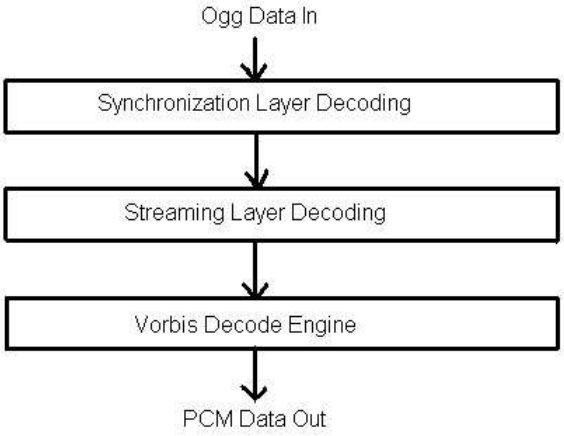


Fig. 1. The decoding layers involved in decoding an Ogg file

Vorbis decoding takes place in two phases - the codec setup phase and the audio decode phase. The first few packets, called headers, contain information necessary for codec setup. Once the codec is setup, decode of audio packets can begin. The headers, in the correct order of occurrence, are as follows:

1) *Identification header* is used to identify bit stream, Vorbis version, sample rate and number of channels
2) *Comment header* includes user text comments and a vendor string for the application/library that produced the bit stream
3) *Setup header* includes extensive codec setup information as well as the complete VQ and Huffman codebooks needed for decoding.

These headers are usually encapsulated in the first couple of Ogg pages. The Ogg pages that follow contain the audio packets. If the audio packet arrives before codec setup is complete it renders the stream undecodable. The algorithm for audio packet decode is given below [2].

1) Decode packet type flag and mode number
2) Decode window shape (long windows only)
3) Decode floor. Decode residue into residue vectors
4) Generate floor curve from decoded floor data
5) Compute dot product of floor and residue, producing audio spectrum vector
6) Inverse monolithic transform of audio spectrum vector, always an MDCT in Vorbis I
7) Overlap/add left-hand output of transform with right-hand output of previous frame
8) Store right hand-data from transform of current frame for future lapping
9) If not first frame, return results of overlap/add as audio result of current frame

III. Implementation

The Ogg-Vorbis decoder is implemented on ADSP-21364 EZ-KIT [6]. The floating point reference code [7] is very generic in nature. Removal of functional redundancy of this decoder is necessary for the entire program to be accommodated in the program memory section (seg_pmco) of the processor [8]. A direct implementation on the processor causes runtime errors due to heap overflow. The default heap size allocated on ADSP-21364 is 0.5 Mbits [8], whereas the dynamic memory requirement estimate for the reference decoder is 2.8 Mbits.

SHARC processors have 32 bit internal memory architecture [9]. Data types like char, short and double are all treated as 32 bit wide data types. If a byte is stored in a 32 bit location the higher 24 bits remain vacant, hence wasted. The reference decoder assumes byte addressable memory and allocates heap space for byte arrays resulting in wastage of 75% of byte-allocated memory. Naturally, byte packing - which obtains byte addressability in word addressable memory - is a solution. It is obtained by changing all byte pointers to custom structure pointers. This structure contains two fields - first, the word address of the 32 bit location in which the byte is present and second, the position of the byte in that particular word. A set of 'wrapper' functions for byte read/write and byte-address change provide the necessary shift from word addressability to byte addressability. Although byte packing reduces the heap requirement, it does not completely satisfy the requirements of the reference decoder. The ADSP-21364 has 3 Mbits of memory split into four memory banks - two 1 Mbit banks and two 0.5 Mbit banks [9]. The *heap* is located in the 1 Mbit memory bank so the single heap size can be increased to a maximum of 1 Mbits using the Linker Description File (LDF) [10], [8]. Byte packing does not reduce the heap requirement below 1 Mbits.

The codec setup phase of Vorbis decoding is memory constrained whereas the audio decode phase is time constrained. Hence a trade-off exists between the memory and execution time (MIPS). The memory requirements can be reduced at the cost of increasing the MIPS count of the decoder. Since ADSP-21364 can operate at a maximum of 333 MHz, it provides enough room for increasing the MIPS count while reducing heap requirements [11].

Vorbis uses a codebook read in vector context [2]. In the reference implementation, the vector tables are built and saved in the heap during codec setup. The heap memory requirement can be reduced by building the vector tables in the real time audio decode loop instead of the codec setup (initialization) phase. In this approach the required vector tables are built in the audio decode phase and the memory is freed immediately. This results in a reduction in the memory requirements of the floating point decoder with file input-output (IO), enabling it to be implemented on the ADSP-21364 without multiple heaps.

A. Real-time decoding

Real time implementation requires a systematic removal of file IO functions and introduction of IO buffers in the reference code. One input buffer and two output buffers are used. The DMA is used for data transfer between the IO buffers in the processor's internal memory and the external peripherals on the EZ-KIT Lite board. The real time implementation uses the AD1835A codec and the 1 MB parallel flash memory on the kit [6]. The flash memory has to be loaded with Ogg data

before decoding can begin. The codec chip contains stereo DACs which convert PCM data to analog audio.

The parallel port DMA (PPDMA) is associated with the input buffer and the serial port DMA (SPORT) with the two output buffers. The PPDMA reads Ogg data from the flash and writes it into the input buffer. The data in the input buffer is decoded and written into the output buffers in a ping-pong fashion. The ping-pong buffer system involves two buffers, A and B, initialized in internal memory. When the serial port (SPORT), which performs the DMA transfer to AD1835A, is reading from buffer A the decode engine should write into buffer B and vice versa. The reading of the audio samples should be slower than the writing of the audio samples to the buffer for the application to be executed in real-time. If this condition is not satisfied then the SPORT routes out old data repeatedly and it appears as if the audio is being played with bursts of repetition. If the writing of the audio samples to the output buffer A is faster than the reading of the samples from the buffer B, then the decode engine has to wait (execute NOP or run another application in case of multitasking) until the reading of buffer B is complete. Once the reading is complete and SPORT starts reading buffer A, and the decode engine starts writing the audio samples into the buffer B.

Care should be taken to map the output buffers optimally to the memory banks as multiple memory banks in ADSP-21364 can be accessed in parallel using the multiple buses [9] and allocation of two frequently accessed sections to the same memory bank results in processor pipeline stalls. Hence the output buffers should be placed in a memory bank which does not contain the program memory section or the regular heap section.

B. Efficient Huffman tree representation

During the setup phase the Huffman trees are to be built using the codebooks as shown in Fig.2. Huffman trees are to be built offline for each of the codebooks and stored in the heap. A Huffman tree is different from a normal binary tree as only the leaf nodes contain values and the non-leaf nodes contain only the child addresses. A data structure, for a node of the tree, having three fields - *value, left child address, right child address* would result in very high memory requirements, as a minimum of three 32-bit words have to be allocated for each node. Therefore, for efficient decoding, a low memory representation of the Huffman tree is needed - a data structure that provides all the properties of the *tree*, and yet consumes considerably less space as compared to the traditional representation. A Compressed-Array-Representation (CAR) of the tree as shown in Fig.3 provides a viable solution. Fig.3 shows the logical and physical representation of the CAR. By performing inorder traversal of the logical tree (i.e. parent - left child - right child at each node recursively) and placing the nodes in the array in the same order as they were traversed *(a, b, d, e, j, l, m, k ⋯)* we obtain compressed array representation. Every node in a tree is a 32-bit integer. For the non-leaf nodes, the higher 16 bits contain the left child address and the lower 16 bits contain the right child address. The leaf nodes contain a 31 bit *value* in the lower 31 bits and a 1 bit leaf-node marker in the most significant bit position. This representation is suitable because normally the trees never overflow the 15 bit index space available and hence can handle trees with less than 2^{15} nodes. In the case of Ogg Vorbis floating point decoder, the tree depth was well within the range.

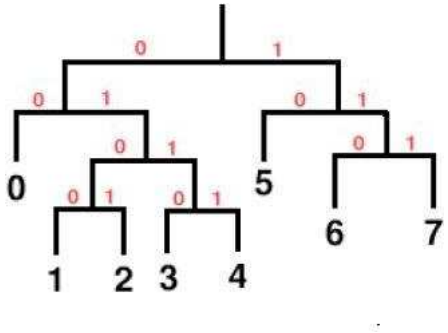


Fig. 2. A Huffman tree

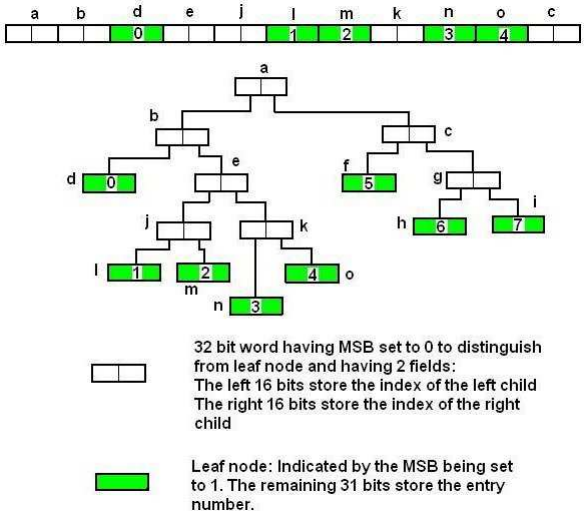
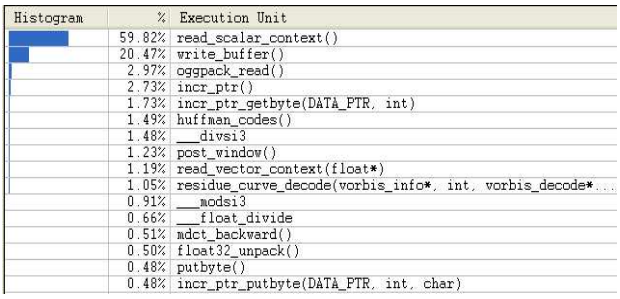


Fig. 3. Efficient CAR representation

IV. Results

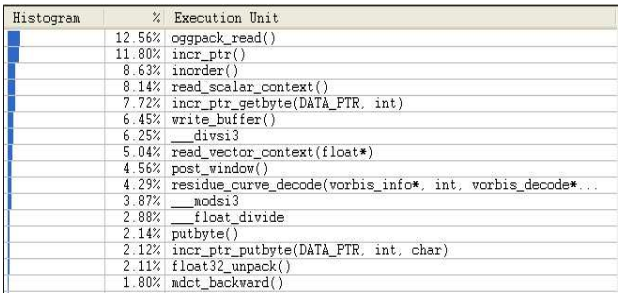
The Ogg-Vorbis decoder is successfully implemented on ADSP-21364 EZ-KIT by optimising the memory usage and computational requirements and the results obtained are promising. Byte packing in synchronization layers resulted in reduction in heap usage by 4000 words. Byte packing in the streaming layer resulted in an additional reduction of 2000 words. Hence the total reduction in heap usage through byte packing is around 6000 words. Shifting the vector table decode from the setup phase to the real time decode loop results in reduction of the required heap memory to about 0.7 Mbits. The memory utilization of different sections is given below.

- 73% of 1 Mbits heap space in Block-1 RAM
- 33% of 0.25 Mbits stack space in Block-3 RAM
- 55% of 0.67 Mbits program space in Block-0 RAM



Histogram	%	Execution Unit
	59.82%	read_scalar_context()
	20.47%	write_buffer()
	2.97%	oggpack_read()
	2.73%	incr_ptr()
	1.73%	incr_ptr_getbyte(DATA_PTR, int)
	1.49%	huffman_codes()
	1.48%	___divsi3
	1.23%	post_window()
	1.19%	read_vector_context(float*)
	1.05%	residue_curve_decode(vorbis_info*, int, vorbis_decode*...
	0.91%	___modsi3
	0.66%	___float_divide
	0.51%	mdct_backward()
	0.50%	float32_unpack()
	0.48%	putbyte()
	0.48%	incr_ptr_putbyte(DATA_PTR, int, char)

Fig. 4. The results of statistical profiling using linear search for traversing huffman trees



Histogram	%	Execution Unit
	12.56%	oggpack_read()
	11.80%	incr_ptr()
	8.63%	inorder()
	8.14%	read_scalar_context()
	7.72%	incr_ptr_getbyte(DATA_PTR, int)
	6.45%	write_buffer()
	6.25%	___divsi3
	5.04%	read_vector_context(float*)
	4.56%	post_window()
	4.29%	residue_curve_decode(vorbis_info*, int, vorbis_decode*...
	3.87%	___modsi3
	2.88%	___float_divide
	2.14%	putbyte()
	2.12%	incr_ptr_putbyte(DATA_PTR, int, char)
	2.11%	float32_unpack()
	1.80%	mdct_backward()

Fig. 5. The results of statistical profiling after using the CAR representation for Huffman trees

- 60% of 0.5 Mbits data memory in Block-2 RAM

Initially the decoder required about 290 MIPS when ported onto the processor. The computational intensity estimate for each of the sub-routines of the real time decoder was done using the VDSP++ environment's statistical profiling tool [12]. The results of profiling are shown in Fig.4

From profiling it was apparent that the function *read_scalar_context()* consumed majority of the computing time. This function traverses the huffman tree with an input code and reads out the value of the leaf node attained. This traversal is the critical section of the code. The Compressed Array representation optimises this critical section by yielding much faster traversal as the tree traversal is reduced to seeking an index in an array. The introduction of CAR reduced the MIPS requirement from 290 MIPS to 80 MIPS. The results of statistical profiling after the CAR implementation is shown in Fig.5. The output was bit exact with the output of the reference codec implemented on Visual C++ environment. The output audio quality for the test audio streams was good and provides a basis for development of an Ogg-Vorbis player using ADSP-21364. A detailed comparison of audio quality of Ogg Vorbis with codecs like MP3, AAC etc has been undertaken by Xiph.org foundation [13].

ACKNOWLEDGMENT

The authors gratefully acknowledge Analog Devices University Program and DSP Applications group, Analog Devices India Product Development Centre, Bangalore for their support in this work.

REFERENCES

[1] Ogg documentation. [Online]. Available: http://www.xiph.org
[2] Vorbis I specification document. [Online]. Available: http://www.xiph.org
[3] A.Kosaka, S.Yamaguchi, H.Okuhata, T.Onoye, and I. Shirakawa, "VLSI implementation of ogg vorbis decoder for embedded applications," in *Proc. IEEE ASIC/SOC Conference 2002*, Sep. 25–28, 2002, pp. 20–24.
[4] H. Karnhall, "Decoding Ogg Vorbis audio with the C6416 dsp using a custom made MDCT core on FPGA," Master thesis, Linkping Institute of Technology, Sweden, 2007.
[5] Tremor reference decoder. [Online]. Available: http://www.xiph.org/vorbis
[6] *ADSP-21364 EZ-KIT Lite Evaluation System Manual Rev 2.0.* Analog Devices Inc, 2005.
[7] Libvorbis reference implementation. [Online]. Available: http://www.xiph.org/vorbis
[8] *ADSP-2136x SHARC Processor Programming Reference Rev 1.0.* Analog Devices Inc, 2005.
[9] *ADSP-2136x SHARC Processor Hardware Reference Rev 1.0.* Analog Devices Inc, 2005.
[10] *Understanding and Using Linker Description Files, Engineer-to-Engineer Note EE-69*. DSP Applications group, Analog Devices, 2005.
[11] *ADSP-21364 SHARC Processor Data Sheet Rev 1.0.* Analog Devices Inc, 2005.
[12] *VDSP++ 4.5 C/C++ Compiler and Library Manual for SHARC Processors Rev 6.0.* Analog Devices Inc, 2006.
[13] Ogg vorbis versus other codecs. [Online]. Available: http://www.xiph.org/vorbis/listen.html