

# On Speed Optimization of MPEG-4 Decoder for Real-Time Multimedia Applications

Gunnar Hovden  
Santa Clara University  
Computer Engineering Department  
500 El Camino Real  
Santa Clara, CA 95053, USA  
ghovden@scudc.scu.edu

Nam Ling  
Santa Clara University  
Computer Engineering Department  
500 El Camino Real  
Santa Clara, CA 95053, USA  
nling@scu.edu

## Abstract

*In this paper we present several speed optimizing techniques in our MPEG-4 video decoder for real-time multimedia applications. Accessor functions and simple data structure are used. Use of pointers are minimized. Fast algorithms and fixed point arithmetic are applied whenever possible. Memory accesses are kept to the minimum. Our results show that our software can decode and display MPEG-4 QCIF videos about 8 times faster than the existing MoMuSys decoder.*

## 1. Introduction

The MPEG-4 standard [1, 2] offers new video tools for coding and decoding of video objects for natural and synthetic videos. MPEG-4 has wide applications, including Internet video and wireless communication. In many applications, real-time high speed decoding is very desirable. The MPEG-4 reference decoder, the MoMuSys decoder [5], is slow and cannot decode and display videos at an adequate speed on the general purpose hardware available today. Its typical decoding speed is 8-10 frames per second for QCIF videos. Higher speed can be obtained in a number of different ways, among them are faster or more specialized hardware architectures, higher clock rates, parallel processing [6], better algorithms and more efficient design. In this paper, we present several of our speed optimization techniques in realizing our MPEG-4 software video decoder.

## 2. Speed optimization techniques

Instead of decoding videos frame by frame, our MPEG-4 decoder decodes one object at a time VOP by VOP. A Video Object Plane (VOP) is a video object in a particular instance of time. In the simplest case of rectangular shaped VOPs, a VOP is equivalent to a frame. Similar to I-, P- and B-pictures in MPEG-1 and MPEG-2, there are three types of VOPs in MPEG-4, namely I-, P- and B-VOPs. An I-VOP (Intra-VOP) is decoded using information from within itself. A P-VOP (Predicted-VOP) is decoded using information from previous I- and P-VOPs in addition to information from within itself. To decode a B-VOP (Bidirectional-VOP), in addition to the information above, information from future I- and P-VOPs is used.

Our decoder is written C, compiled using GNU C-compiler, Version 2.8.1. All tests are performed on a Hewlett Packard C240 workstation (236MHz, 256 Mbyte RAM), running HP-

UNIX Version 10.20. The code for displaying the decoded video on the screen is written using function calls to XWindows libraries.

Our program is simple and uses accessor functions, rather than complicated data structures as in [5]. We avoid long sequences of pointers that requires several memory accesses. Accessor functions give sufficient functionality. In [5], each VOP is represented as a matrix of pointers to macroblocks, each of which is a matrix of pointers to four blocks. Our VOPs are stored as matrices with the actual pixel values (three matrices, one for each of the three components  $Y$ ,  $C_r$  and  $C_b$ ). The pixels can be accessed through a set of accessor functions that write and read data to and from the VOP as blocks or macroblocks. The disadvantage of our data structure is that it requires the data to be moved to and from the VOP whenever it is to be modified on a block or macroblock basis, but this does not happen very often. Our advantage with fewer pointers is that it reduces the need to access different memory areas.

In our program, all arithmetic in the decoding process is carried out using integer arithmetic or using integers that simulate fixed point arithmetic to reduce computation complexity due to floating point numbers. Fast algorithms (e.g. fast IDCT) are used everywhere possible. Low level function calls are used extensively. Code segments for copying or moving data (like a matrix) that are normally written using a single or a nested loop are replaced with calls to low-level functions like `memset()` and `memcpy()`. This produces code that is harder to read, but the increase in speed is noticeable. The number of calls to `malloc()` and related functions (for memory allocation) has been kept to an absolute minimum, since these are functions that may not have efficient implementations in the operating systems where the program may be executed. This also results in less memory fragmentation.

### 3. Performance

Table 1 shows the performance of our decoder, measured in frames per second, for several different bitstreams in two formats (QCIF and SIF), both with rectangular shaped VOPs and with arbitrary shaped VOPs. Speed (with display) is the speed for decoding when the result is displayed on the screen. Speed (no display) is the speed for decoding when the frames are not shown or stored anywhere. The bitstreams are generated using the MoMuSys encoder [5]. In the I-VOPs, each VOP is intra-decoded within itself, P-VOPs are predicted from earlier VOPs and motion compensation techniques are therefore necessary.

**Table 1. Decoding speed in frames per second for different movies in different formats**

File	Frame size	Number of frames	Types of VOPs	Shape of VOP	Speed (with display) [frames/sec]	Speed (no display) [frames/sec]
foreman.qcif	176x144	200	1 I, 199 P	rectangular	39.0	59.3
foreman.qcif	176x144	200	1 I, 9 P <sup>*</sup>	rectangular	36.9	55.9
child.sif	352x240	300	1 I, 9 P <sup>*</sup>	rectangular	12.2	19.1
child.sif	352x240	300	300 I	arbitrary	9.4	13.2
claire.qcif	176x144	126	1 I, 9 P <sup>*</sup>	rectangular	50.2	92.0
claire.qcif	176x144	126	1 I, 125 P	rectangular	53.9	101.6
claire.qcif	176x144	126	1 I, 9 P <sup>*</sup>	arbitrary	42.7	64.0
suzie.qcif	176x144	150	1 I, 9 P <sup>*</sup>	rectangular	43.4	67.1

\* pattern is repeated

Table 1 shows that bitstreams with arbitrary shaped VOPs [3] are more time consuming to decode than bitstreams with rectangular VOPs. This is due to the complexity of shape decoding that involves context-based arithmetic decoding [4]. Also, bitstreams with only P-VOPs after the first necessary I-VOP is decoded faster than bitstreams that includes one I-VOP for every nine P-VOPs. The reason is that the blocks contained in macroblocks that have no prediction error in P-VOPs do not need further decoding, while in the case of an I-VOP every block need to be decoded using Huffman decoding and IDCT (Inverse Discrete Cosine Transform).

#### 4. Complexity analysis

The UNIX tool `prof` from GNU is used to profile the decoder. The result is summarized in Table 2. The different modules in Table 2 perform the following tasks:

**Display frame:** Displaying decoded frames on an XWindows display.

**Decode block:** AC/DC prediction, reordering of coefficients (zig-zag, etc.) and dequantization.

**Access VOP:** The accessor functions for reading/writing blocks and macroblocks to/from the VOP.

**Matrix manipulation:** Matrix adding, subtracting, dividing, multiplying, transposing and copying.

**Motion compensation:** Motion compensation.

**Compose:** Copying the part of an arbitrary shaped VOP that are inside the shape, to the frame to be viewed.

**Other:** Routines that do not belong to any other modules.

**Input/Output:** Reading the bitstream from disk, and performing double buffering on bit level.

**Pixel interpolation:** Interpolation of pixels for  $C_r$  and  $C_b$  components for regular motion compensation and for  $Y$ ,  $C_r$  and  $C_b$  components for half pixel motion vectors.

**Rectangular padding:** Padding of rectangular VOPs [7].

**Variable Length Decoding:** Huffman decoding.

**Saturation control:** Limiting the decoded pixel values to the range [0, 255].

**IDCT:** Fast Inverse Discrete Cosine Transform (8x8).

**Arbitrary padding:** Padding of arbitrary VOPs [7].

**Shape decoding:** Decoding of shapes, including the use of context-based arithmetic decoding [4].

Two representative test files have been used, `foreman.qcif` and `claire.qcif`. Both have frame size of 176x144 pixels. The video `foreman.qcif` is coded with rectangular shaped VOPs, while `claire.qcif` is coded with arbitrary shaped VOPs. Both files are coded with the repeating pattern of one I-VOP followed by nine P-VOPs. Table 2 shows several interesting results. The routine that displays the frames takes about one third of the total time. It may be possible to decrease this time. Decode block is the single part that takes the most of the decoding time for rectangular shaped VOPs. This part requires less time in arbitrary shaped VOPs because blocks outside the coded object are not coded and need not be decoded. The second most time consuming part for the arbitrary shape decoding is padding. Padding is necessary for motion compensation. Pixels that are not part of an object, and hence are not coded, need to be padded with values so that they can be used as references for macroblocks when doing motion compensation. Again this routine makes extensive use of low level function calls to efficiently fill memory areas with the padded pixel values, but the algorithm itself is costly to implement. IDCT, which used to be one of the most time consuming parts in older MPEG-1/MPEG-2 decoders, only accounts for 0.3% to 0.4% of the total time in our software. The reason for this is the added complexity of other parts in the decoding scheme. Naturally, no time is spent on

arbitrary shape padding and shape decoding for the bitstream with rectangular shape, since there is no shape information for rectangular VOPs.

**Table 2. Percentage of time spent in each module for two different files**

Module	Rectangular shaped VOPs (1 I, 9 P) [%]	Arbitrary shaped VOPs (1 I, 9 P) [%]
Display frame	34.1	33.2
Decode block	16.9	8.5
Access VOP	13.4	14.1
Matrix manipulation	6.8	2.0
Motion Compensation	6.7	3.5
Compose	4.3	4.5
Other	3.8	8.5
Input/Output	3.5	3.3
Pixel interpolation	3.4	2.3
Rectangular padding	2.7	0.0
Variable Length Decoding	2.7	1.1
Saturation control	2.4	2.7
IDCT	0.3	0.4
Arbitrary padding	0.0	9.7
Shape decode	0.0	6.1
Total	100.0	100.0

## 5. Conclusion

The software is currently able to decode and display videos in QCIF format (176×144 pixels) in real time (37-54 frames per second). Videos in CIF (352×288 pixels) and SIF format (352×240 pixels) are currently decoded fast enough (9-12 frames per second) for real-time applications like video phones and surveillance cameras. Our decoder is about 8 times faster than the MoMuSys decoder [5].

## 6. References

- [1] ISO/IEC JTC1/SC29/WG11, N2202, Information Technology - Coding of Audio-Visual Objects: Visual ISO/IEC 14496-2, Committee Draft, Tokyo, March 1998.
- [2] ISO/IEC JTC1/SC29/WG11, MPEG98/M3100, MPEG-4 Video Verification Model Version 9.1, February 1998.
- [3] J. Ostermann, E. S. Jang, J. Shin and T. Chen, "Coding of Arbitrarily Shaped Video Object in MPEG-4", IEEE International Conference on Image Processing, 1997, pp. 496-499.
- [4] N. Brady, F. Bossen and N. Murphy, "Context-based Arithmetic Encoding of 2D Shape Sequences", International Conference on Image Processing, 1997, pp. 29-32.
- [5] ISO/IEC JTC1/SC29/WG 11, N2205, Text of 14496-5 (MPEG-4 simulation software) final committee draft, March 1998.
- [6] Y. He and I. Ahmad, "A software-Based MPEG-4 Video Encoder Using Parallel Processing", IEEE Transactions on Circuits and Systems for Video Technology Vol. 8, No. 7, Nov. 1998, pp. 909-920.
- [7] W. Chen, C Gu and M. C. Lee, "Repetitive and Morphological Padding for Object-based Video Coding", International Conference on Image Processing, 1997