H.26X 인코딩 속도향상 구현

이승현, 강의선, 강석찬, 윤성규, 임영환
숭실대학교 컴퓨터학과

Implementation of a H.26X Software Encoding Speed up Research

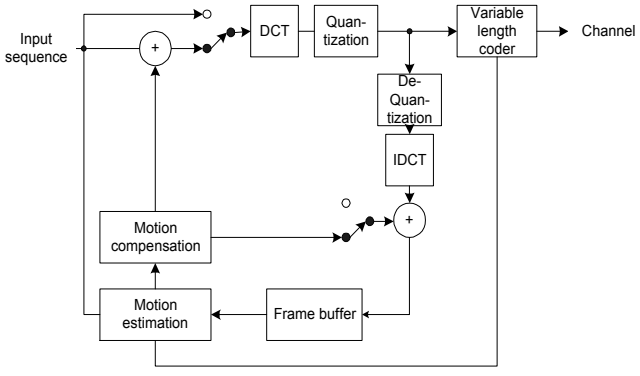
seung-hyeon Lee, Eui-Sun Kang, Seock-Chan Kang, Sung-Kyu Yoon , Young-Hwan Lim
Dept.of Computer Science Soongsil University

요약

본 논문에서는 기존의 H.26X 부호화 시의 속도 저하의 원인인 움직임 추정의 높은 복잡도를 개선하여 소프트웨어만으로 실시간 부호화를 하는 방법을 모색하고자 한다. 논문에서는 스탠포드 대학에서 작성한 소프트웨어 부호기를 분석하여 속도를 높이기 위한 해결 방법으로 두 가지 방법으로 접근하였다. 첫째로 부호기의 속도 저하의 원인인 움직임 추정의 시간을 줄이기 위하여 현재 많이 사용되고 있는 4 단계 탐색 알고리즘을 개선한 새로운 알고리즘을 제안하였다. 둘째로 현재 작성된 코드를 인텔사에서 제공하는 MMX^{TM} 명령어를 사용, 병렬처리 하여 속도 향상을 꾀하였다.

1. INTRODUCTION

1980 년대 후반 이후 좁은 대역폭의 전통적인 전화선에서 화상 전화로 많은 요구가 있었다. JPEG 알고리즘을 발전시킨 비디오-폰(video-phone), 비디오-회의(video-conference)를 위한 국제 표준이 개발되었는데 그것이 H.261 과 H.263 이다.



[그림-1] 비디오 부호기(Encoder)

비디오 압축은 디지털 비디오 신호의 저장과 전송을 위해 발전되어 왔지만 비디오를 실시간으로 압축하기 위해서는 하드웨어의 도움을 받았었다. 하드웨어로 구현된 실시간 부호기(COpressor/DECompressor:CODEC)를 소프트웨어로 구현하기 위해서 현재 스탠포드 대학에서 구현하여 놓은 H.261 부호기(p64)를 구현 하였으나, 소프트웨어로 구성된 부호기는 초당 너무 작은 양의 프레임 생성을 하였다.

COMPRESSION	MOPS
RGB to YCbCr conversion	27
Motion estimation (exhaustive search for p=8)	608
Inter-/Intraframe coding	40
Loop filtering	55
Pixel prediction	18
2-D DCT	60
Quantization and zig-zag scanning	44
Entropy coding	17
Frame reconstruction	99
TOTAL	968

[표-1] H.261 부호기의 MOPS 의 요구사항

위의 [표-1]은 MOPS(Million Operation Per Second)라는 단위로 CIF 형식의 30 프레임/초 일반적인 H.261/H.263 의 MOPS 의 요구사항을 보여주고 있다. [표-1]에서 볼수 있듯이 움직임 추정(Motion Estimatio)알고리즘의 높은 복잡도((High complexity)는 다른 알고리즘에 비하여 엄청난 시간이 할애되는 것을 알수 있었다. 따라서 우리가 접근하여야 할 부분이 잠정 결정되었고, 좀더 구체적으로 Mircosoft Visual Studio 6.0 에 있는 Profile 을 사용하여 스탠포드 대학에서 구현하여 놓은 H.261 을 함수별로 처리 시간을 분석하여 확인하여 보았다.

아래 [표-2]는 기본적인 p64 프로그램내의 함수별 분석 도표로서 원래의 소스의 시간 소요 순위 함수별 실행 결과이다. [표-2]에서 FastBME 라는 함수가 움직임 추정(Motion Estimation)을 수행하는 함수이고, 전체 프로그램

의 67.1%의 실행시간을 차지함을 보여주고 있다.

FUNCTION TIME	%	FUNC+CHILD TIME	%	HIT COUNT	FUNCTION
126282.8	63.7	126282.8	63.7	5960150	_SAD_MACROBLOCK (ME.OBJ)
18518.3	9.3	18518.3	9.3	486486	_mwtell (stream.obj)
6686.5	3.4	132969.3	67.1	34650	_FastBME (me.obj)
5824.8	2.9	5824.8	2.9	351	_VerifyFiles (p64.obj)
5670.2	2.9	5670.2	2.9	1053	_LoadMem (mem.obj)
4743.5	2.4	4743.5	2.4	208494	_ChenIDct (chendct.obj)
4142.8	2.1	4142.8	2.1	208494	_ChenDct (chendct.obj)
3776.8	1.9	23703.3	12.0	208494	_EncodeAC (codec.obj)
3003.8	1.5	3003.8	1.5	208494	_CCITTFlatQuantize (transform.obj)
2965.9	1.5	2965.9	1.5	1505350	_mputv (stream.obj)
1963.8	1.0	3823.0	1.9	1150786	_Encode (Huffman.obj)

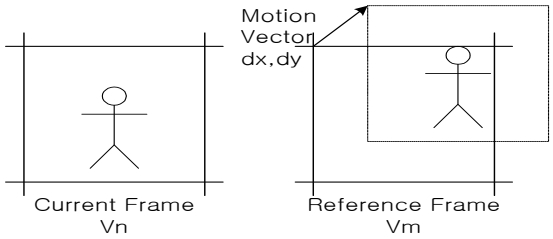
[표-2] p64 분석(350 프레임)

또한, FastBME 함수 자체는 3.4%만을 차지하고 있음을 알 수가 있고 움직임 추정시 높은 점유율을 보이는 이유는 하위의 함수 SAD_Macroblock 이라는 함수가 오래 걸려 실질적인 속도 저하의 원인이 되는 것이다. 이는 하위 함수 호출 회수를 [표-2]로 보아도 직관적으로 알 수 있다.
위의 [표-1]과 [표-2]의 분석을 토대로 다음과 같은 방법으로 접근하고자 한다.

첫째. 빠른 부호화를 위한 적은 연산의 새로운 알고리즘 구현
둘째. 새로운 명령어(MMX™)를 통한 빠른 속도 구현

2. 빠른 부호화를 위한 적은 연산의 새로운 알고리즘 구현

움직임 추정은 현재의 프레임과 참조 프레임간의 움직임을 비교하는 것이다..



[그림-2] 블록 정합(block matching)

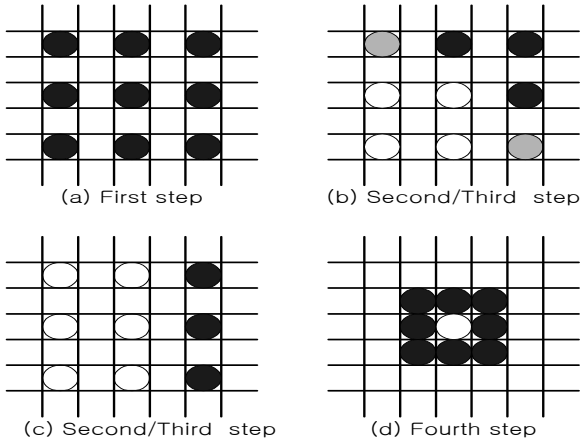
나누어진 각 프레임의 사각 블록에 의해 블록 정합 알고리즘(block matching algorithms:BMA)은 참조 프레임의 탐색원도우 안쪽에서 현재 프레임과 정합(maching)되는 블록(block)을 찾는다. 프레임간의 움직임 추정(ME)을 하기 위해 많은 고속 블록 정합 알고리즘(Fast Bolck Matching Algorithm:FBMA)이 제시되고 있고, 연산의 복잡도 역시 감소에 목적을 둔다. 아래의 [표-3]은 지금까지 개발되어 있는 고속 탐색 알고리즘(Fast-search algorithm)들이다.

Three-step search Algorithm
2D-logarithmic search Algorithm
Orthogonal search Algorithm
Cross search Algorithm
New three-step search Algorithm
Block-based gradient descent search Algorithm
Four-step Search Algorithm (4SS)
Center-biased Orthogonal Search Algorithm (CBOSA)
Hybrid Adaptive Search Algorithm (HASA)
Hierarchical Partial Distortion Search Algorithm (HPDS)
Hybrid Search Algorithm (HSA)

[표-3] 고속 탐색 알고리즘((Fast-Search Algorithms)

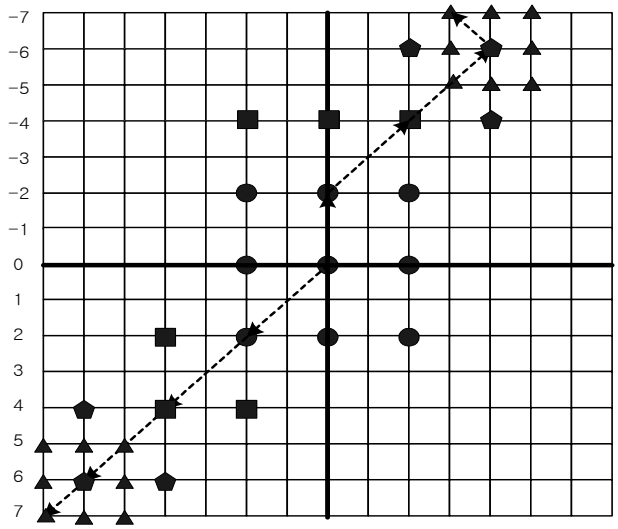
이러한 여러 가지 방법들 중에서 새로운 3 단계 탐색 알고리즘(New Three-Step Algorithm : NTSS)과 4 단계 탐색 알고리즘(Four-Step Algorithm : 4SS)이 많이 알려져 있다. 4SS는 최근에 제안되어 뛰어난 화질과, 적은 계산량으로 많이 알려진 방법이다.
그래서, 4 단계 알고리즘을 분석한 결과, 4 단계 알고리즘을 개선하여 더욱 적은 탐색 점을 사용할 수 있음을 발견하였다.

개선에 대한 패턴은 아래의 [그림-3]에서 설명되어지고, 전체적인 알고리즘은 [그림-4]에서 설명되어지고 있다. 전체적인 알고리즘은 4 단계 알고리즘과 흡사하다. 그러나 4 단계 탐색에서 두 번째와 세 번째 단계에서 탐색하는 패턴의 수는 모서리일 경우 3 개의 탐색 점을, 모퉁이일 경우는 5 개의 탐색 점을 가지는데, 여기서 모퉁이의 5 개의 탐색 점을 2 개를 줄였다. 4 단계 탐색 알고리즘과의 패턴 비교는 아래의 [그림-3]과 같다. 그림에서 검은 점은 4SS 에서만 탐색하는 점이다. 따라서 (b)의 패턴이 블록에 2 번이 나오게 되는 경우 4 개의 점을 줄일 수 있다. 2,3 단계는 (b)또는 (c)의 패턴이 나오기 때문에 매 블록마다 최대 4 개의 비교 점이 줄게 된다.



[그림-3] 새로운 4 단계 알고리즘의 탐색 패턴

초기 탐색 크기(initial step size)는 4 단계 탐색 알고리즘(Four-Step Search Algorithm)과 마찬가지로 최대 움직임 치환 d 의 4 분의 1 이다. 작은 initial step size 로 인해 확률적 탐색(4SS)알고리즘도 d=7 인 탐색윈도우의 경계에 도달할 4 단계의 탐색 단계를 필요로 한다. [그림-4]에서 표현하는 것은 탐색 2 가지 방법으로 새로 적용한 알고리즘으로 찾을 때의 상황을 표현한 것이다. 좌측 상단의 경우 (9+3+3+8)=23 회의 점을 필요로 한다. 우측 하단의 경우, d=7 일 때 알고리즘에서 가장 좋지 않은 경우로 (9+3+3+8)=23 회의 체크 포인트를 필요로 한다. 최대 23 의 탐색 점을 필요로 한다. 작은 움직임의 경우에서는 4 단계 탐색(4SS)과 같은 탐색 점(checking point)을 필요로 한다.

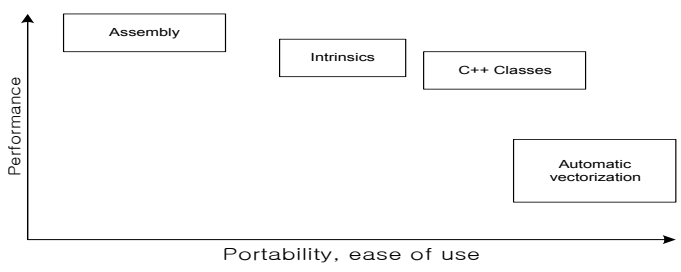


[그림-4] 새로운 4 단계 탐색(N4SS)

더욱이, 탐색 단계의 최소 BDM 대조 점이 가운데이면, 단계의 크기는 반으로 줄고, 4 단계로 뛰어 넘는다는 것은 4SS 와 동일하다.

3. MMX™ 코드에 의한 속도 향상

MMX™ 의 장점은 여러 연산을 동시에 수행하는데 있다. 따라서 반복적인 계산에 적용을 하면 많은 연산의 부하를 줄일 수 있을 것이다.



[그림-5] 방법간의 성능과 이식성의 비교

MMX™ 테크놀러지의 프로그래밍 방법은 여러 가지 있다. Intel 에서 제공하는 The Intel C/C++ Compiler 를 사용하는 방법과 어셈블리로 프로그래밍 하는 방법이다. [그림-5]에서 각각의 성능과 이식성에 대한 비교표이다. The Intel C/C++ Compiler 는 Intrinsic 라는 MMX™ 명령어를 제공하여 컴파일을 하면 어셈블리로 생성을 해주는 편리한 기능이 있다. 그러나 [그림-5]에서 보여주듯이 어셈블리로 직접 프로그램밍 하는 것 보다 성능 면에서의 차이를 나타내주고 있다. 따라서 MMX™ 이 필요한 부분을 직접 어셈블리로 작업하였다.

3.1 Cost –Function(SAD Macroblock)의 코드화

블록 정합 알고리즘(BMA)은 탐색창안 후보 움직임 벡터(CMV-Candidate motion vectors)를 찾는데 탐색창안에서 후보 움직임 벡터를 찾으려면 각 벡터의 값을 계산하여야 하는데, 블록의 차이를 계산 방법을 cost-function 이라 한다. Cost-function 의 종류는 다음의 [표-4]와 같다.

The Mean-squared Difference (MSD)
The Mean Absolute Difference (MAD)
The Cross-Correlation Function (CCF)
The Pixel Difference Classification (PDC)
The Sum of Absolute Difference (SAD)

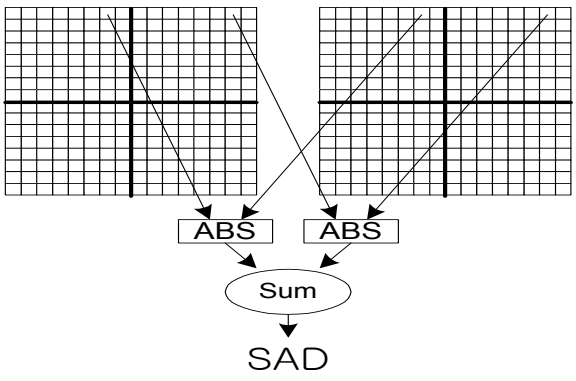
[표-4] Cost function 의 종류

대부분 절대 오차의 합(Sum of Absolute Difference : SAD)을 많이 사용하고 있는데, H.261/H.263 의 코드에서도 C 로 구현이 되어 있다. 속도를 개선하기 위해서 C 로 구현되어 있는 SAD 를 MMX™ 코드로 변환하였다.

$$SAD = \sum_{i=0}^{15}\sum_{j=0}^{15}\left|Block_{ref}[i][j] - Block_{pred}[i][j]\right|$$

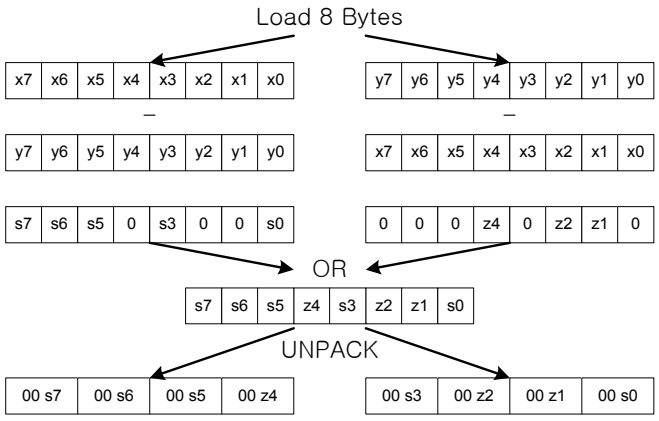
[수식-1] Sum of Absolute Difference(SAD)

SAD 의 수식을 보면 알 수 있듯이 한 프레임과 비교프레임에 대해 각각 하나의 픽셀들에 절대 오차(Absolute Difference)를 구한 뒤 모두 더하는 것이다.



[그림-5] SAD 작동 방법

위에 [그림-5]에서는 절대 오차의 합(SAD)을 구하는 방법을 보여주고 있다. 그림에서 보여주듯이 절대 오차의 합(SAD)을 계산하기 위해 많은 픽셀들의 절대값을 구하는 루틴들의 반복을 볼 수가 있다.



[그림-6]ABS 구현 다이어그램

[그림-6]은 각 MMX™ 레지스터에서 절대 오차를 구하는 루틴에 대한 방식을 다음의 그림에서 보여 주고 있다. 데이터는 부호가 없는 형식을 사용하여 각각의 값을 감(減)한 뒤, 양수의 값만을 취하는 방식을 사용하여 속도를 향상을 이루었다. 최종 향상은 구현 결과에서 보기로 한다.

3.2 FastBME

앞장에서 언급한 알고리즘(Fast Block Motion Estimation)을 구현한 부분이다. 여기서 SAD_Macroblock 도 호출을 한다. Motion Vector 을 구한 후 몇 가지의 변수를 계산하는데, 여기서 많은 시간이 걸려, FastBME 에서도 SAD_Macroblock 과 마찬가지로 블록을 모두 계산을 하여서 많은 처리 시간이 걸리게 된다. 제한된 레이스터을 효율적으로 다음 [그림-7]처럼 사용하였다.

mm0: esi
mm1: edi
mm2:
mm3:
mm4: MWOR
mm5: VAROR
mm6: 0x0000000000000000
mm7: VAR

[그림-7]MMX™ 레지스터

3.3 DCT 코드화

FastBME 다음으로 처리속도에 많은 부분을 차지한 함수가 DCT 와 IDCT 함수이다. 현재 p64 에서 사용하는 함수는 ChenDCT 방법이다. 효율적인 DCT 의 계산을 위해 여러 가지 알고리즘들이 제안이 되었다.

The Mean-squared Difference (MSD)
The Mean Absolute Difference (MAD)
The Cross-Correlation Function (CCF)
The Pixel Difference Classification (PDC)
The Sum of Absolute Difference (SAD)

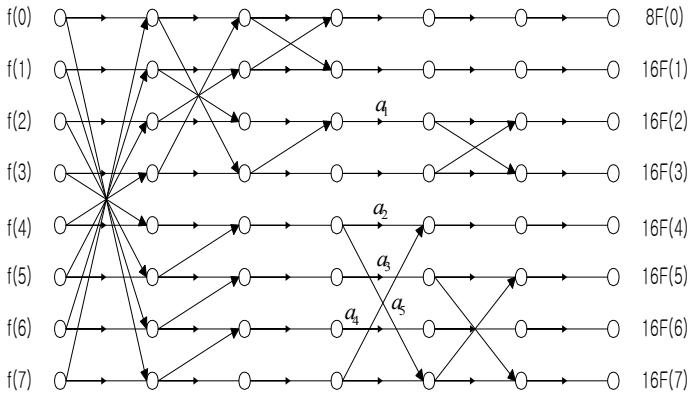
[표-5]DCT 의 알고리즘

이 중 AAN 알고리즘은 8x8 블록마다 postscale 을 위한 64 개의 곱셈 연산과 80 개의 곱셈연산(MACs:multiply-accumulates)과 464 개의 덧셈 연산을 필요로 한다. DCT, IDCT 알고리즘은 다음과 같다.

$$F(u,v)=\frac{c(u)}{2}\frac{c(v)}{2}\sum_{i=0}^{7}\cos\left[\frac{(2i+1)}{16}u\pi\right]\sum_{j=0}^{7}Xij\cos\left[\frac{(2j+1)}{16}v\pi\right]$$

$$c(k)=1, iffk\neq 0, \frac{1}{\sqrt{2}}otherwise$$

[수식-2] DCT



a_1=0.707 a_2=0.54 a_3=0.707 a_4=1.30 a_5=0.38

[그림-8]AAN 알고리즘

DCT 를 수행 시 8 개를 묶음으로 해서 1 차원 DCT 를 2 번만에 수행을 하여, MMX™ 의 최대 효율을 나타내도록 한다. MMX™ 는 정수연산만을 지원하는 단점이 있다. 이것을 처리하는 방법으로 곱해지는 상수들을 일정량을 곱해서 계산한 뒤 다시 나누는 방법을 사용하면 된다.
최종 향상은 구현 결과에서 보기로 한다.

4. 구현 결과

본 논문의 결과는 다음과 같은 환경에서 도출하였다.

O S : Microsoft Windows 98
C P U : Intel® Pentium® II Processor 233MHz

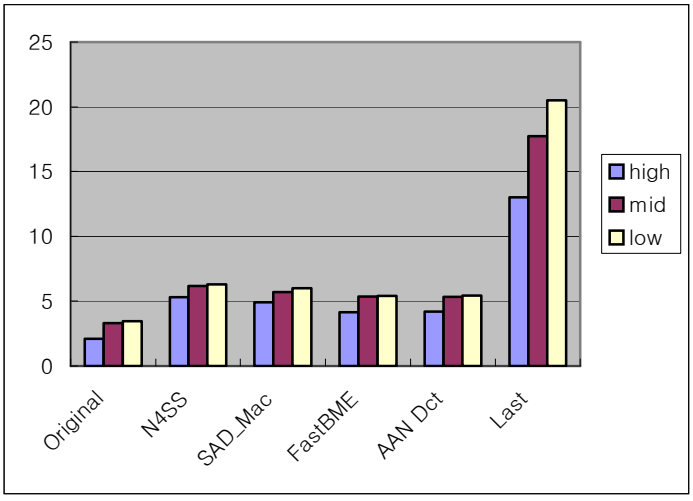
R A M : 64MByte

H.261 을 구현한 p64 는 초당 평균 2.9 프레임이 생성이 되는데 새로운 알고리즘을 적용한 결과 5.9 프레임이 나왔다. 다음의 [표-7]은 부분별 생성한 프레임이다. 오른쪽의 괄호 안의 숫자는 원래의 코드를 1 로 보았을 때 향상된 프레임의 비율이다. 움직임이 많은 프레임이 6.17 배의 향상이 이루어 졌고, 움직임이 적은 경우는 초당 20 프레임이 생성되었다. 전체적으로 평균 5.78 배의 향상이 되었고, 초당 17.074 정도의 프레임이 나왔다.

	High	Mid	Low	Average
Original	2.10 (1)	3.31 (1)	3.44 (1)	2.95 (1)
N4SS	5.29 (2.51)	6.16 (1.86)	6.30 (1.82)	5.91 (2)
SAD_Mac	4.91 (2.33)	5.70 (1.72)	6.00 (1.74)	5.54 (1.87)
FastBME	4.14 (1.97)	5.36 (1.62)	5.41 (1.57)	4.97 (1.68)
AAN DCT	4.20 (1.99)	5.32 (1.61)	5.42 (1.57)	4.98 (1.68)
Last	13.00 (6.17)	17.71 (5.35)	20.50 (5.95)	17.07 (5.78)

[표-6]전체 성능 향상 결과

위의 표를 도식화 한 그림이다.



[그림-9]전체 성능 향상 결과

5. 결론

일반적으로 비디오를 인코딩하기 위해서는 하드웨어를 사용하였으나 컴퓨터의 성능이 좋아짐에 따라, 소프트웨어만으로 실시간 부호화를 할 수 있게 되었다. 본 논문에서는 부호화 시에 속도 저하의 가장 큰 원인이 되는 이유를 분석하여 MMX™ 로 코드를 변환하였다. 또한 부호화 시에 가장 많은 부분을 차지하는 움직임 추정을 개선하기 위한 방법으로 새로운 4 단계 알고리즘을 개선하여 속도 향상을 이루었다. 이제 일반화가 된 MMX™ 로 접근하여 어느 컴퓨터에서라도 쉽게 부호화를 할 수 있게 되었다. H.26X 에 맞추어 이 논문이 쓰여졌지만, 다른 동영상도 같은 방법으로 작동이 되므로, 여러 가지의 형태의 소프트웨어 부호기가 만들어질 수 있을 것이다. MPEG 의 재생이 소프트웨어로 하는 것이 자연스러운 지금과 같이, 비디오 부호화도 소프트웨어로 하는 것일 일반화 될 것이다.

6. Reference

1. ITU-T Standardization Sector of ITU, “Video Coding for Low Birate Communication,” Draft ITU-T Recommendation H.263 Version 2, September 1997.
2. ITU-T Standardization Sector of ITU, “Video Coding Test Model Near-Term, Version 8(TMN8), Release 0,” H.263 Ad Hoc Group, June 1997.
3. Borko Furht, Joshua Greenberg and Raymond Westwater, Kluwer Academic Publishers, “Motion Estimation Algorithms for Video Compression”, 1997
4. Borko Furht, Joshua Greenberg and Raymond Westwater, kluwer Academic Publishers, Motion Estimation Algorithms for video Compession, 1997
5. K.R.Rao, J.J. Hwang, Techniques & Standards for Image. Video & Audio Coding, Prentice Hall, 1996
6. Cote, G., Gallant, M. and Kossentini, F. "Efficient Motion Vector Estimation and Coding for H.263-Based Very Low Bit Rate Video Compression." Online document available at URL http://www.ece.ubc.ca/spmg/, 1997.
7. CCITT H.263-Image Compression - Online document available at URL http://www.stud.ee.ethz.ch/~rmprince/h263.html
8. Performance Analysis of Intel MMX Technology for an H.263 Video Encoder - Ville Lappalainen