논문 2007-44SD-10-6

이동형 디지털 방송을 위한 H.264/AVC 디코더 시스템의 구현 및 성능 분석

(Implementation and Performance Analysis of H.264/AVC Decoder System for Mobile Digital Broadcasting)

정 진 원*, 송 용 호**

(Jinwon Jung and Yong Ho Song)

요 약

멀티미디어 비디오 응용의 이용이 증가함에 따라 모바일 임베디드 시스템 환경에서 H.264/AVC 기반의 디코더 시스템 구현에 대한 수요가 증가하고 있다. H.264/AVC 디코딩 작업은 내부적으로 많은 연산을 필요하므로, 임베디드 시스템 환경 및 멀티미디어 비디오 응용의 기대 품질에 따라 다양한 구현 방법이 사용될 수 있다. 하지만, 주어진 모바일 임베디드 시스템 환경에 적합한 구현 방법을 선택하기 위해서는 임베디드 시스템의 연산 능력과 비디오 디코딩 작업에 필요한 연산 요구량에 대해 정확한 분석이 필요하다. 본 논문에서는 모바일 임베디드 단말 환경을 위한 H.264/AVC 디코더의 하드웨어 및 소프트웨어 구현 방안을 제안하고, 이에 대한 성능 측정 방법 및 결과를 제시하였다. 또한 리눅스 기반의 모바일 임베디드 시스템에서의 비디오 디코딩 시스템의 성능 제한 요소를 판별함으로써 효과적인 구현 방법을 보이고 있다.

Abstract

The increasing demand on the use of multimedia video contents drives more mobile embedded systems to incorporate H.264/AVC decoding capability. An H.264/AVC decoder often requires high computation bandwidth during its decoding phase. Depending upon processor computation capability and multimedia contents complexity, the decoder can be implemented either in hardware or software. However, without a thorough analysis on the performance and resource requirements, it is difficult to choose a cost-effective methodology of implementing this codec. This paper presents both hardware and software implementation of H.264/AVC decoding subsystem in mobile embedded systems, and quantitatively analyses the performance and resource requirements. It also shows the methodology to identify performance bottleneck in Linux-based mobile embedded systems, which is in turn used to select feasible and efficient implementation methodology.

Keywords : H.264/AVC, Performance Analysis, Decoder, Firmware, Embedded Systems

* 학생회원, ** 정회원, 한양대학교 대학원 전자컴퓨터 통신공학과
(Dept. of Electronics and Computer Engineering, Hanyang Univ.)
※ 본 논문은 정보통신부의 출연금 등으로 수행한 정보통신연구개발산업의 연구결과입니다.
※ 본 논문의 내용을 발표할 때에는 반드시 정보통신부 정보통신연구개발사업의 연구결과임을 밝혀야 합니다.
※ 본 논문은 2007년도 「서울시 산학연 협력사업」의 「나노 IP/SoC설계기술혁신사업단」의 지원으로 이루어졌습니다.
접수일자: 2007년5월26일, 수정완료일: 2007년9월28일

I. 서 론

최근 영상 전화, PMP (Portable Media Player), 내비게이터 등 동영상 재생 기능을 갖춘 임베디드 단말기의 사용이 점차 증가하고 있다. 이와 함께 고품질 비디오에 대한 사용자의 요구가 증가하고 있어, 임베디드 시스템에서도 고화질 비디오 재생 기능의 구현에 따른 필요성이 증가하였다. 이에 따라 고화질과 고압축율을 지원하는 H.264/MPEG-4 AVC[12](이하 H.264)가 다양한 이동형 디지털 방송(DMB, DVB-H), HDTV, 및 IP-TV 등 동영상 스트리밍 서비스 등에서 표준 비디오

코덱으로 사용되고 있다.[5]

H.264 코덱은 비디오 디코딩 단계에서 많은 연산 작업을 필요로 한다. 특히 시스템 자원이 제한적인 모바일 임베디드 환경에서는 자원의 성능 및 가용성, 운영체제 오버헤드 등 시스템 관련 요소와 화면의 크기, 초당 프레임 수 등 비디오 스트림 관련 요소에 의해 H.264 코덱의 성능이 직접적으로 영향을 받게 된다. 따라서 주어진 임베디드 환경에 최적화된 비디오 재생 기능을 구현하기 위해서는 타깃 플랫폼의 구조, 자원, 성능 등에 관해 철저히 분석해야 하며, 또한 디코딩 작업의 필요 연산량에 대한 세밀한 분석이 수행하여야 한다. 그리고 분석 결과를 바탕으로 주어진 임베디드 환경의 코덱 수행 능력을 판단하고, 필요할 경우 하드웨어 코덱 등과 같은 추가의 연산 기능을 제공할 필요가 있다.

본 논문에서는 ARM9을 기반으로 하는 모바일 임베디드 환경에서 H.264 디코딩 시스템의 하드웨어 및 소프트웨어 구현 방법에 따른 프로세서 연산량 및 메모리 대역폭 등에 관한 성능 분석 사례를 제시한다. 또한 성능 측정 과정에서 임베디드 환경 내부의 동작에 대한 방해(Intrusiveness)를 최소화하기 위해 방안에 대해 제안한다. 이를 위해 먼저 하드웨어 코덱과 소프트웨어 코덱을 이용하여 각기 다른 두 개의 H.264 디코딩 시스템을 구현하였다. 그리고 구현 방법에 따른 성능 및 자원 사용량에 대해 분석 작업을 수행하였다. 디코딩 시스템의 구현 시 운영체제의 오버헤드를 측정하기 위하여 ARM linux 환경과 펌웨어 환경에서 H.264 디코딩 시스템을 구동시켜 성능을 비교 분석하였다.

본 논문의 구성은 다음과 같다. Ⅱ장에서는 관련 연구에 대해 설명하고, Ⅲ장에서는 본 논문에서 사용하는 H.264 표준에 대해 간략하게 정리한다. Ⅳ장에서는 하드웨어 및 소프트웨어를 이용한 H.264 디코딩 시스템의 구현 결과에 대해 설명하고, Ⅴ장에서는 논문에서 제안하는 성능 측정 방법에 관해 기술한다. 그리고 Ⅵ장과 Ⅶ 장에서는 각각 ARM linux 환경과 펌웨어 환경에서의 H.264 디코더의 성능 측정 결과를 분석하고, 마지막으로 Ⅷ 장에서 결론을 맺는다.

Ⅱ. 관련 연구

H.264 디코더의 성능을 분석하는 방법에는 분석 모델(Analytic Model)을 이용한 방법[1], 시뮬레이션을 이용한 방법[3], 가상 플랫폼을 이용한 방법[2], 그리고 실제 시스템에서 측정하는 방법[21] 등이 있을 수 있다. 분석 모델 방법에서는 성능 계산에 필요한 이론적인 공식과 인자를 산출한 다음 구현하고자 하는 시스템과 관련된 인자(parameter)값을 대입하여 성능을 구하게 된다. 또한 시뮬레이터나 가상 플랫폼을 사용하는 경우에는 SimpleScalar-ARM 등과 같은 ISS (Instruction Set Simulator) 또는 트랜잭션 수준의 모델 (Transaction-Level Model)을 이용한 가상 플랫폼이 사용된다. 이러한 방법들은 제한된 시간에 H.264 디코더의 성능을 상대적으로 쉽게 예측할 수 있으나, 시뮬레이션 모델의 구체성, 테스트 벤치마크의 적합성 등에 따라 실제 임베디드 환경과 성능 상의 오차를 발생시킬 수 있다.

실제 시스템을 구현하여 성능을 측정하는 방법이 가장 정확성을 높이는 방법이지만, 측정 방법에 따라 여전히 오차가 발생할 가능성은 존재한다. 예를 들어, ARM 프로세서 기반의 임베디드 시스템에 H.264 디코더를 구현하는 경우 프로세서 내부에 제공되는 ETM(Embedded Trace Macrocell)[9]을 이용하여 프로세서의 실행 트레이스를 수집하고 이를 분석함으로써 디코더의 성능을 측정할 수 있다. 하지만, ETM에서 수집되는 트레이스의 길이가 충분치 않아 전체 디코더의 동작에 대한 시스템 성능을 분석하는 것은 용이하지 않다. 이 경우 성능 측정에 관련된 소프트웨어 코드를 이용하게 되며, 이 코드가 디코더에 대해 원치 않는 오버헤드를 발생시킬 수 있어 성능 측정 결과의 정확도를 떨어트리는 결과를 낳는다.

이 논문에서는 소프트웨어 및 하드웨어로 디코더를 구현하는 경우, 디코딩 동작에 미치는 오버헤드를 최소화하면서 디코더 내부 모듈의 동작 시간에 대한 측정 방법을 제시한다. 그리고 ARM920T[9, 13] 기반 임베디드 환경에서 동영상 재생기를 구현하고, 성능에 직접적인 영향을 미치는 임계 함수(critical function)의 추출 방법을 제시한다.

Ⅲ. H.264/MPEG-4 AVC 개요

H.264는 기본적으로 비디오 프레임의 효과적이고 강력한 압축 및 전송을 위해 설계되었다[7]. 응용분야에는 양방향 비디오 통신(화상회의 또는 화상전화), 방송 또는 고화질 비디오를 위한 압축 그리고 패킷 네트워크(packet network)를 통한 비디오 스트리밍 서비스 등이 포함된다.

H.264는 사용하는 슬라이스와 코딩 방법에 따라 Baseline, Main, Extended 프로파일로 나누어진다.(표 1

표 1. H.264/AVC Baseline, Main 및 Extended Profile.[13]
Table. 1. H.264/AVC Baseline, Main and Extended Profile.

	Baseline	Main	Extended
I and P Slices	YES	YES	YES
B Slices	NO	YES	YES
SI and SP Slices	NO	NO	YES
Multiple Reference Frame	YES	YES	YES
In-Loop Deblocking Filter	YES	YES	YES
CAVLC Entropy Coding	YES	YES	YES
CABAC Entropy Coding	NO	YES	NO
Flexible Macroblock Ordering (FMO)	YES	NO	YES
Arbitrary Slice Ordering (ASO)	YES	NO	YES
Redundant Slices (RS)	YES	NO	YES
Data Partitioning	NO	NO	YES
Interlaced Coding (PicAFF, MBAFF)	NO	YES	YES

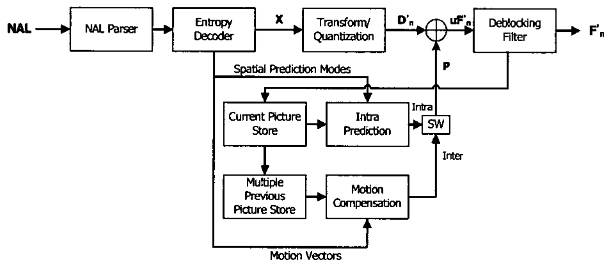

그림 1. H.264/AVC 디코더의 내부 모듈 동작
Fig. 1. Internal operation of H.264/AVC decoder.

참고) 각 프로파일의 차이에 따라 응용분야가 달라지는데, Baseline 프로파일의 응용분야로는 화상전화, 화상회의 그리고 무선 통신이 있고, Main 프로파일은 TV 방송과 비디오 저장, 그리고 Extended 프로파일은 스트리밍 미디어 응용분야에 특히 유용하다.[14]

그림 1은 H.264 디코더의 내부 모듈들의 동작 과정을 보이고 있다. 우선 H.264 디코더는 NAL (Network Abstract Layer) 단위로 이루어진 비트스트림을 입력받아서 개별 NAL을 분리한다. 분리된 데이터들에 대해 엔트로피 디코딩을 수행하여 계수 X를 생성한다. 생성된 계수들은 역양자화되고 다시 역변환되어 오차블록 $D'n$을 생성한다. 또한 H.264 디코더는 입력받은 비트스트림으로부터 디코딩된 헤더 정보를 사용하여 Intra Prediction과 Motion Compensation을 거쳐 예측 블록 P를 생성하는데, P는 다시 $D'n$과 더해져 $uF'n$을 생성한다. 생성된 $uF'n$은 Deblocking Filter를 거쳐 디코딩된 블록 $F'n$을 생성한다. 이 $F'n$은 저장되어 다음 예측 블록을 생성하기위해 사용된다.

IV. 하드웨어 및 소프트웨어 디코더 시스템의 구현

1. 타겟 플랫폼

본 논문에서는 구현하는 임베디드 H.264 디코더 시스템은 크게 ARM920T 기반의 임베디드 플랫폼(그림 2의 우측 보드), VideoLAN의 VLC 멀티미디어 비디오 재생기(이하 VLC)[19], 그리고 H.264 디코더(그림 1)로 구성된다. 하드웨어 디코더 시스템은 모바일 H.264 하드웨어 디코더 칩(Samsung S3CA480[11])을 장착한 개발 보드(그림 2의 좌측 보드)로 구현되며, 소프트웨어 디코더 시스템은 동일 디코딩 기능을 VLC 내부에 소프트웨어 모듈로 구현하였다. ARM920T에서 동작하는 비디오 재생기는 비디오 데이터를 디코딩하기 위해서 하드웨어 개발 보드에 구현된 H.264 디코더 칩을 구동하거나, 혹은 내부 소프트웨어 코덱을 사용한다. 하드웨어 디코더 시스템을 구성하는 두 개의 보드는 커넥터를 이용하여 주소와 데이터 및 인터럽트 신호를 서로 연결하며, ARM920T 보드에 장착되어 있는 LCD 화면을 이용하여 동영상 디코딩 결과를 확인할 수 있다.

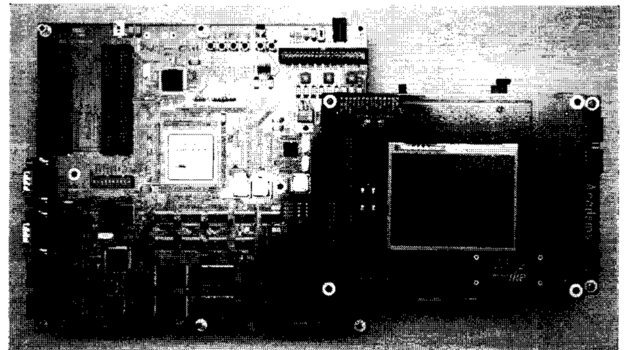

그림 2. H.264/AVC 임베디드 시스템을 위한 타겟 플랫폼
Fig. 2. Target platform for H/264/AVC embedded systems.

2. 멀티미디어 비디오 재생기

본 실험에서 구현한 멀티미디어 비디오 재생기는 지상파 DMB 재생기의 비디오 비트스트림을 해석하고 이를 재생한다. 이를 위해 ARM linux 환경에서 VLC 재생기에 DMB 비트스트림에 대한 demux 기능을 추가하였다. VLC는 오픈 소스 멀티미디어 재생기이며, 필요에 따라 코드의 수정이 가능하다. 이 재생기는 다양한 네트워크 스트림과 MPEG-2 TS (Transport Stream) 시스템을 내장 지원하기 때문에 이를 기반으로 하여 지상파 DMB 등 이동형 방송을 위한 단말 응용의 구현이 비교적 용이하

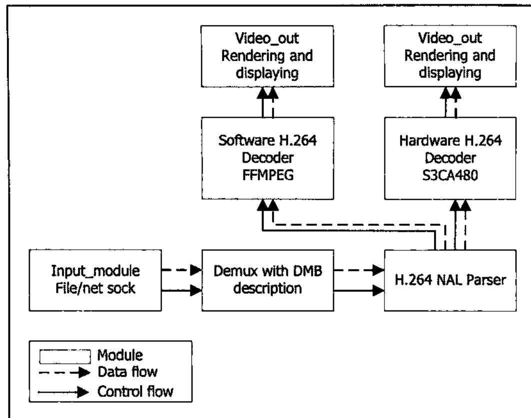

그림 3. VLC에서 H.264 처리 과정
Fig. 3. Processing of H.264 stream in VLC.

다. 본 논문에서는 지상파 DMB의 베이스밴드 모뎀을 사용하여 비디오 스트림을 수신하는 대신, VLC의 기능을 확장하여 네트워크를 통한 지상파 DMB 스트림의 수신이 가능하도록 하였다.

확장된 VLC는 지상파 DMB의 비디오 스트림의 처리를 위해 소프트웨어와 하드웨어 H.264 디코더를 사용하였다. H.264 소프트웨어 디코더로는 FFMPEG[20] 라이브러리를 사용하고 하드웨어로는 S3CA480 멀티미디어 코프로세서[11]를 사용했다. 각각의 디코더는 동일한 구조의 재생기 내에서 서로 배타적으로 동작하기 때문에 실행 시 서로 영향을 받지 않는다.

그림 3은 VLC에서 H.264 스트림을 처리하는 과정을 도시한 것이다. 지상파 DMB 스트림을 위한 demux를 통해 생성된 H.264 스트림은 NAL Parser를 통해 NAL 단위로 분리되고, VLC의 컴파일 옵션에 따라 소프트웨어 또는 하드웨어 디코더에 의해 처리된다.

하드웨어 H.264 디코더로 사용되는 S3CA480은 그림 4와 같은 구조로 MCU와 연결되어 있으며 VLC에서는 소프트웨어 디코더와 동일한 방법으로 호출된다. 하드웨어 디코더의 경우 FPGA 칩을 통해서 MCU와 연결되기 때문에 실제 동작 시에는 FPGA 칩을 통해서 S3CA480에 구동에 관련된 신호를 전송하게 된다.

리눅스 환경에서 VLC에 하드웨어 H.264 디코더의 구동 관련 API를 제공하기 위해 디바이스 드라이버를 제작 및 사용하였다. 이 API를 VLC에 적용해서 하드웨어 디코더를 사용할 경우 이전의 소프트웨어 디코더 대신 동작하게 된다.(그림 3 참고)

S3CA480는 H.264 디코더 외에 LCD 제어기 및 TV 출력 기능 등을 포함하고 있다. S3CA480의 H.264 디코더는 그림 5에 도시된 바와 같이 크게 Parsing 부분과 Codec

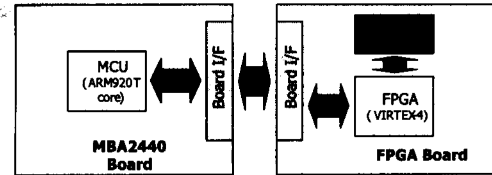

그림 4. S3CA480을 이용하기 위해 FPGA 보드와 연결한 MCU 보드
Fig. 4. Interface between FPGA and MCU board for the use of S3CA480.

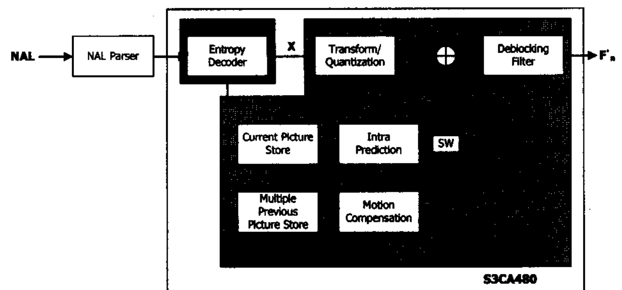

그림 5. 하드웨어 H.264 디코더의 Block Diagram
Fig. 5. Block diagram for hardware H.264 decoder.

부분으로 나누어져 있다. Parsing는 H.264 비트스트림의 엔트로피 디코딩을 수행하고, Codec은 그 외 역양자화 및 역변환, Intra Prediction, Motion Compensation과 Deblocking Filter 등의 기능을 담당한다. S3CA480은 내부에 H.264 디코딩 동작을 수행에 필요한 메모리를 포함하고 있으며, 메인 메모리에 저장된 H.264 비트스트림을 내부 메모리에 복사하여 디코딩하며, 그 결과를 다시 내부 메모리에 저장한다. 저장된 결과는 자체 LCD 제어기를 통하여 출력되거나, 호스트 프로세서가 읽어 후처리 작업을 진행할 수 있다.

V. H.264/MPEG-4 AVC 디코더의 성능 측정 방법

이 장에서는 ARM linux 및 펌웨어 환경에서 소프트웨어 및 하드웨어 H.264 디코더 시스템의 성능을 측정하는 방법을 제시한다. 제시한 방법은 소프트웨어 및 하드웨어 디코더에 동일한 방법으로 적용하여 성능 측정 결과를 얻을 수 있다.

1. ARM linux 환경에서의 성능 측정

ARM linux에서 제공하는 시스템 함수 중에서 gettimeofday()는 현재 시각 값을 μsec 단위로 반환해 준다. 특정 함수의 실행 시간을 측정하고자 할 때, 함수 호출 부분과 반환 부분에서 gettimeofday()를 호출하고,

그 값의 차이를 계산한다. (그림 6 참고) 하지만 측정하려는 함수 또는 모듈이 호출되는 횟수가 많아질수록 이런 gettimeofday() 함수가 실행되는 시간이 누적되어 시간 측정 결과에 대한 정확도가 떨어진다. 그 예로 H.264 디코더에서 320x240 크기의 포맷을 갖는 비디오 프레임을 하나를 처리할 때, 16x16 크기의 매크로블럭으로 이루어진 이미지를 처리하기 위해 매크로블록 처리함수를 총 300회 호출한다. 이 경우 gettimeofday()가 호출되는 횟수는 총 600회가 된다. 그리고 H.264 디코더의 성능을 실측하기 위해 실제 디코딩 동작과는 상관이 없는 함수를 사용하기 때문에 그로 인한 캐쉬 메모리 내부에 오염(pollution)이 발생하게 되고, 이로 인한 오버헤드가 실측 결과의 정확성을 떨어뜨린다.

실제로 gettimeofday() 함수만을 이용해서 소프트웨어 H.264 디코더의 성능을 실측했을 때, 단일 프레임의 평균 처리시간이 137.49ms인 것에 반해 각각의 모듈들의 평균 처리 시간을 합한 값은 156.72ms로 측정되어, 약 19ms 정도의 오차가 발생했다. 이것은 디코더의 내부 모듈의 사용 빈도에 따라 성능 측정을 위한 함수들의 호출 빈도 역시 증가하기 때문에, 그것으로 인한 오버헤드가 누적되기 때문이다.

위와 같은 이유로 이 논문에서는 H.264 디코더가 구동되기 전에 캐쉬 라인에 gettimeofday() 함수를 lockdown 해서 디코딩 과정에 성능 측정으로 인한 캐쉬 오염을 제거한다. 그리고 성능 측정에 사용하는 gettimeofday() 함수의 latency를 구해서 측정 결과에 그 값을 반영하였다. 이와 같은 방법으로 H.264 디코더의 성능 측정 과정에서 발생하는 오버헤드를 최소화하여 측정 결과의 정확성을 높일 수 있다.

```
struct timeval module_start;
struct timeval module_end;

int (*funcptr)(struct timeval *tv, struct timezone *tz);

funcptr = gettimeofday;

CacheLockdown(funcptr, size_of_funcptr);

......

{
        gettimeofday(&module_start, NULL);
        ......
        decode_module();
        ......
        gettimeofday(&module_end, NULL);
}
```

그림 6. 캐쉬 lockdown과 gettimeofday()함수를 이용한 H.264 디코더 모듈의 time complexity 측정 예

Fig. 6. An example of time complexity measurement for H.264 decoder module using cache lockdown and gettimeofday() function.

그 외에 성능 측정 과정에서 결과를 저장하기 위해 발생하는 메모리 및 파일 I/O가 발생한다. 이것 역시 측정 과정의 오버헤드가 되기 때문에, 결과의 정확성이 떨어지는 원인이 된다. 그래서 이 논문에서는 함수의 성능을 측정하기 전에 시스템 메모리의 heap 영역에 충분한 양의 메모리를 미리 할당해서 gettimeofday() 결과를 반환하는 위치로 사용하고, 모든 동작이 완료된 후에 그 결과를 File I/O의 형식으로 출력하도록 하였다. 위와 같은 방법으로 H.264 디코더의 주요 동작들을 측정함으로써, I/O과정 때문에 발생할 수 있는 오버헤드를 제거할 수 있었다.

앞서 설명한 방법은 ARM linux의 애플리케이션 레벨에서 성능 측정을 목적으로 했을 때 적용할 수 있는 방법이다. 이 논문에서 사용한 하드웨어 H.264 디코더의 경우 커널 레벨에서 디바이스 드라이버를 통해 제어하는데, gettimeofday() 함수의 경우 커널 레벨에서는 지원하지 않기 때문에 이 방법을 그대로 적용하기에는 무리가 있다. 커널 레벨에서는 성능 측정에 사용한 gettimeofday() 함수의 커널 버전인 do_gettimeofday()를 사용하였다. 그리고 마찬가지로 미리 할당한 메모리 영역에 측정 결과를 반환한 다음 모든 동작이 종료된 후에 출력하도록 하였다.

2. 펌웨어 환경에서의 성능 측정

ARM linux 환경과는 다르게 펌웨어 환경에서는 시스템을 유지하기 위한 백그라운드 프로세스들이 존재하지 않는다. 따라서 측정하고자하는 동작 외에는 성능에 영향을 줄 수 있는 부분이 감소하게 된다. 하지만 펌웨어 환경에서는 O/S에서 지원하는 일반적인 시스템 함수들이 없기 때문에 원하는 기능을 직접 추가해야한다.

이 논문에서는 펌웨어 환경에서 H.264 디코더의 time complexity를 측정하기 위해 watchdog timer를 이용하는 timer 함수를 작성해서 사용했다. 일반적으로 watchdog timer는 임베디드 시스템에서 오류가 발생했을 경우 reset을 위해 사용되지만, 이 경우 펌웨어 환경에서 동작하는 H.264/AVC 디코더의 각 모듈의 성능을 측정하는데 사용했다. 성능 측정에 사용한 watchdog timer는 한 클록 당 16μsec를 나타낸다.

그림 7은 성능 측정을 위해 S3C2440에서 지원하는 watchdog timer의 블록도이다. watchdog timer는

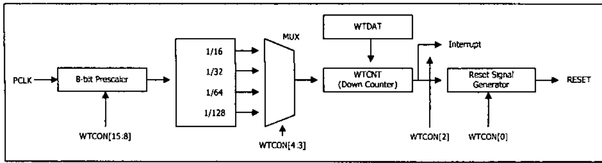

그림 7. Watchdog Timer 블록도[13]
Fig. 7. Block diagram of watchdog timer.

WTCON라는 제어 레지스터를 이용해서 사용할 수 있는데, 이 레지스터는 watchdog timer의 prescale 값과 인터럽트 발생 여부 등을 제어할 수 있다. 이 논문에서는 시스템의 오류 복구가 아니라 성능 측정을 위해 watchdog timer를 사용하므로 down counter가 0이 되어도 인터럽트 신호를 발생하지 않도록 한다.

펌웨어 환경에서도 ARM linux 환경과 마찬가지로 H.264 디코더의 성능을 측정하기 위해 별도의 함수를 사용하기 때문에 watchdog timer로 인한 오버헤드를 측정 결과에 영향을 주지 않기 위해 마찬가지로 캐쉬 lockdown을 사용하였다.

펌웨어 환경에서의 성능 측정도 ARM Linux 환경과 마찬가지로 I/O 가 발생하게 된다. 따라서 이 I/O로 인한 오버헤드를 최소한으로 줄이기 위해서 H.264 디코딩하는 동안 측정 결과를 메모리의 사용하지 않는 영역에 저장해두고, 모든 동작이 끝난 후에 저장한 값을 출력하도록 한다.

펌웨어 환경에서의 성능 측정의 장점은 단일 프로세스로 동작하는 환경에서의 디코더 성능을 O/S 환경에서의 성능과 비교할 수 있는 것이다. 그러기 위해서 대상이 되는 플랫폼의 환경 설정은 ARM Linux 환경과 동일하게 설정되어야 한다. 이 실험에서는 ARM Linux환경과 동일한 동작 주파수와 MMU 설정을 해서 기본적은 동작 환경이 동일하다. 이렇게 측정한 결과를 ARM linux에서 측정한 결과와 비교함으로써 O/S로 인한 오버헤드를 계산할 수 있다.

VI. ARM Linux 환경에서의 성능 측정 결과

1. 소프트웨어 H.264 디코더 측정 결과

표 2는 이 논문에서 제시한 방법으로 소프트웨어 H.264 디코더의 내부 모듈들이 H.264 비트스트림을 처리할 때 걸리는 time complexity를 보여준다. 표의 측정 결과는 15초 분량의 H.264 Baseline 프로파일의 비트스트림을 처리한 시간 중에서 한 개의 프레임을 처리했을 때의 평균값을 보여주고 있다. 이 때 슬라이스의 종류

표 2. 소프트웨어 H.264 디코더의 Time Complexity
Table 2. Time complexity of software H.264 decoder.

Modules	Time (μsec)	Clock Cycle
Entropy Decoder	1,202.85	4,811.40
Transform/Quantization	15,725.65	62,902.60
Intra Prediction	54,331.80	217,327.20
Motion Compensation	38,346.18	153,384.72
Deblocking Filter	38,850.93	155,403.72
Total	148,457.41	593,829.64

는 무시하였다. 표에서의 Total 값은 측정한 H.264 디코더의 내부 모듈들의 time complexity를 모두 합한 값이다.

동일한 H.264 비트스트림의 처리 속도를 내부 모듈들의 합이 아니라 각각의 프레임 단위로 성능을 측정했을 때 time complexity는 하나의 프레임 당 평균 147.3ms으로 표 2의 TOTAL 값과 약 1ms 정도의 오차만이 발생하였다. 이것은 논문에서 제시한 측정 방법의 정확성을 보여준다.

이 논문에서 제시한 방법으로 H.264 디코더의 내부 모듈들을 측정 결과 한 개의 프레임을 처리하는 데 평균 148.5ms가 걸리는데, 측정에 사용한 비디오가 25fps로 인코딩된 것으로 한 개의 프레임을 최대 40ms (1 sec / 25 frames) 내에 처리할 수 있어야 실시간 재생이 가능하다는 것을 감안할 때, 이 실험 환경에서는 실시간 H.264 비디오 출력이 불가능하다는 것을 확인할 수 있다. 또한, 그림 8의 소프트웨어 디코더의 모듈별 처리시간을 백분율로 나타낸 값을 보면, 가장 많은 computing power를 필요로 하는 모듈이 Intra Prediction 인 것을 알 수 있다. 하지만, Intra Prediction 외에도 Motion Compensation과 Delocking Filter 역시 40ms 정도의 처리시간을 필요로 하고 있기 때문에, 3개의 모듈이 critical function이 되고 있음을 알 수 있다.

H.264 Baseline 프로파일에서 사용하는 I 슬라이스와

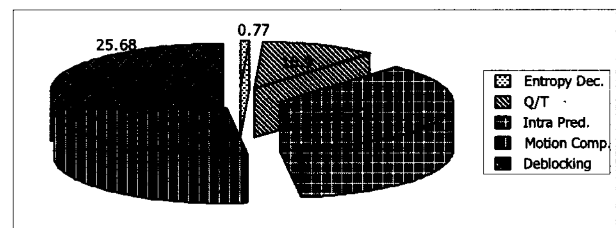

그림 8. 소프트웨어 H.264 디코더의 각 모듈별 Time Complexity의 백분율
Fig. 8. Breakdown of time complexity in software H.264 decoder.

P 슬라이스 디코딩 과정에서 I 슬라이스의 경우는 Motion Compensation 과정이 호출되지 않고, 실험에 사용한 비트스트림의 경우 초당 1개의 I슬라이스를 포함하기 때문에 실제로 Intra Prediction이 호출된 횟수와 Motion Compensation이 호출된 횟수에 차이가 발생한다. 하지만, 여기에서의 time complexity는 한 개의 프레임의 평균 처리 시간을 말하고 있으므로, 이런 호출 횟수에 대한 차이는 무시하기로 한다.

이처럼 소프트웨어 H.264 디코더를 임베디드 환경에서 사용할 때 성능이 저하되는 것은 PC 환경보다 연산 속도가 떨어진다는 것 외에도, 임베디드 환경에서는 별도의 멀티미디어 가속기능이나 DSP 등을 지원하지 않는 것도 원인이 될 수 있다. 그리고 임베디드 환경에서 임베디드 프로세서의 처리 속도와 시스템 버스 및 메모리의 속도 차이 역시 전체 성능의 저하를 발생시킬 수 있다.

이런 성능상의 문제점을 개선하기 위해 ASIP (Application Specific Instruction Processor)[10]를 이용해서 critical function에 대해 최적화된 기능을 제공하는 방법과 하드웨어 디코더로 소프트웨어 디코더를 대체하는 방법 등이 있다. 위의 성능 측정 결과를 보면 대부분의 모듈이 critical function이 되고 있으므로, 이 논문에서는 하드웨어 디코더를 이용해서 동일 기능에 대해 성능을 다시 측정하였다. 이렇게 하드웨어 디코더를 이용해서 H.264 디코더의 성능을 측정한 결과를 다시 위의 결과와 비교함으로써 디코더의 동작 특성을 알 수 있다. 다음 절에서는 S3CA480 multimedia coprocessor를 이용해서 H.264/AVC 디코더를 구현했을 때 소프트웨어 디코더와 동일한 방법으로 성능을 측정하고 그 결과를 비교해 보았다.

2. 하드웨어 H.264 디코더 측정 결과

표 3은 하드웨어 H.264 디코더로 비트스트림을 처리할 때 한 개의 프레임 처리에 걸리는 평균 time complexity를 보여준다. 여기서 사용한 하드웨어 디코더 내부 모듈을 주요 동작에 따라 Parsing 부분과 Codec 부분으로 분리해서 μsec 단위로 측정했다. 이 때 동작 주파수 400MHz이다.

표 3. 하드웨어 H.264 디코더의 Time Complexity
Table 3. Time complexity in hardware H.264 decoder.

Modules	Time (μsec)	Clock Cycle
Parsing	1,010.34	4,041.36
Codec	4,318.13	17,272.52
Host I/F Read	12,945.99	51,783.96
Host I/F Write	379.98	1,519.92
Total	18,654.44	74,617.76

하드웨어 디코더를 이용할 경우 1개의 프레임을 처리하는 데 걸리는 평균 시간은 18.65ms로 25fps의 동영상을 출력하는 데 충분한 성능을 보여주며, 소프트웨어 디코더와 비교했을 때 약 8배의 성능 향상을 보여준다. 모듈별로 비교하면 소프트웨어 디코더의 critical function이었던 Intra Prediction, Motion Compensation, Deblocking Filter가 하드웨어 모듈로 대체되면서 이 부분에 대해서는 약 36배까지 향상된 것을 알 수 있었다. 하지만, S3CA480을 이용한 하드웨어 디코더를 사용하면서 디코딩 동작 외에 데이터 read/write 동작이 발생하게 되는데, 그 이유는 ARM920T가 FPGA 칩을 통해서 하드웨어 디코더를 구동하고 있고, 하드웨어 디코더와 LCD 출력을 별도의 프로세스로 분리해서 동작시키고 있어서 프로세스 간 디코딩 결과(RGB 데이터)에 대해서 별도의 read/write 동작이 필요하기 때문에다.

그림 9는 하드웨어 H.264 디코더의 각 모듈별 Time Complexity 결과를 백분율로 보여주고 있다. 이처럼 하드웨어 H.264 디코더를 적용함으로써 전체 성능은 소프트웨어 디코더를 사용할 때보다 향상되었지만, 자체만을 놓고 보았을 때 read/write 동작이 70% 이상을 차지하는 것을 알 수 있다.

이런 read/write 동작으로 인한 오버헤드는 하드웨어 디코더를 측정한 플랫폼의 구조적인 원인으로 발생한다. V장에서 설명한 것처럼 하드웨어 디코더인 S3CA480은 FPGA 모듈에 직접 연결되어 있기 때문에, MCU 보드의 LCD 제어부와 직접 연결할 수 없다. 따라서 디코딩 결과를 LCD 화면에 출력하기 위해서는 하드웨어 디코더의 결과를 MCU에서 read 해서 LCD의

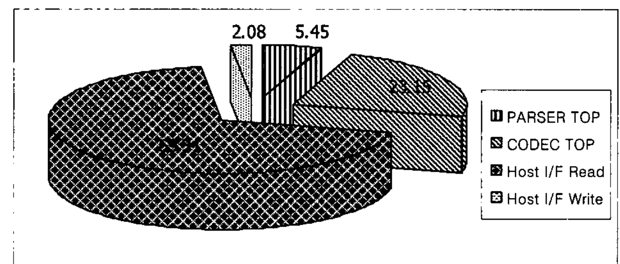

그림 9. 하드웨어 H.264 디코더의 각 모듈별 Time Complexity의 백분율
Fig. 9. Breakdown of time complexity in hardware H.264 decoder.

표 4. 하드웨어 디코더의 LCD controller 사용 여부에 따른 MCU와 S3CA480의 처리시간과 utilization
Table 4. Processing time and utilization of MCU and S3CA480 with and without LCD controller.

(a) without LCD controller

MCU (Host I/F Read/Write)	S3CA480 (PARSING PART, CODEC PART)
13,343.97 μsec (71.39 %)	5,346.47 μsec (28.61 %)

(b) with LCD controller

MCU (Host I/F Write)	S3CA480 (PARSING PART, CODEC PART)
388.98 μsec (6.78%)	5,346.47 μsec (93.22 %)

프레임버퍼로 복사해주어야 한다. 그래서 디코딩 동작과는 별도로 디코딩 결과를 read하는 동작을 수행해야 하는 것이다.

만약 S3CA480에 자체적으로 포함되어 있는 LCD controller를 이용해서 디코딩 결과를 출력한다면, 디코딩 결과를 MCU에서 read하는 동작이 필요 없어지기 때문에 1개의 프레임의 평균 처리 시간은 약 5.73ms가 되고, 이 경우의 MCU와 S3CA480의 utilization을 현재 상태와 비교해보면 표 4와 같다.

표 4.(b)는 H.264 디코딩 결과를 하드웨어 디코더에서 직접 LCD controller를 이용해서 출력할 때의 MCU와 하드웨어 디코더의 utilization을 보여주고 있다. 표 4.(a)에서 MCU에서 Host Interface read/write동작을 모두 수행해야 했던 것과는 달리 하드웨어 디코더에서 직접 처리함으로써 전체 디코딩 수행시간이 줄어든 것을 알 수 있다. 또한 같은 동작을 처리하는데 MCU의 utilization이 65%가량 줄어든 것을 확인할 수 있었다. 이것은 여러 스레드가 동시에 처리되어야 하는 실시간 멀티미디어 재생기를 구현할 때, MCU가 비디오 디코딩 동작 외에 다른 작업을 할 수 있는 여지가 생겼다는 것을 알려준다.

3. 소프트웨어와 하드웨어 H.264 디코더 구현에 필요한 메모리 요구량 비교

H.264 디코더를 소프트웨어 또는 하드웨어를 이용해서 구현할 경우 각각 필요로 하는 최소 메모리의 크기가 달라질 수 있다. 특히 이 논문에서 사용한 S3CA480처럼 내장된 메모리와 버스를 사용할 경우 디코딩 과정에서 사용되는 대부분의 데이터들을 위해 별도로 메모리를 할당할 필요가 없기 때문에 디코더 구현에 필요한 메모리 크기가 매우 작아진다. 동일 동작을 하는 디코더에서 필요한 메모리의 크기가 줄어들게 되면, 임베디드 프로세서에서 메모리로부터 read/write하는 데이터양이 줄어들게 되는데, 이것은 결과적으로 MCU와 시스템 버스 그리고 메모리의 속도로 인한 성능 저하를 감소시키는 역할을 한다.

표 5. H.264 디코더 구현에 필요한 메모리 요구량
Table 5. Memory requirements for the implementation of H.264 decoder.

Buffer Name	S/W Decoder (bytes)	H/W Decoder (bytes)
Reconstruction frame	115,200	115,200
Reference frames	115,200	-
Slice map	1,200	-
Reference indices	1,200	-
Motion vectors	19,200	-
Intra-prediction modes	4,800	-
CBP values	1,200	-
MB types	300	-
QP values	300	-
CAVLC coefficient counts	7,200	-
MB temp data	2,048	-
Constants	2,048	2,048
Total	269,896	117,248

여기서는 H.264 디코더를 소프트웨어와 하드웨어(S3CA480)로 각각 구현했을 때 사용되는 기본적인 변수들을 [1]에서 제시한 방법으로 계산해서 디코더 구현에 필요한 메모리의 요구량을 표 5에 정리하였다. 이 때 계산 기준이 된 이미지의 크기는 320x240이고 reference picture의 개수는 한 개로 하였다. 계산 결과 소프트웨어 디코더를 구현할 때 디코딩에 필요한 메모리가 270KB인 것에 비해 하드웨어를 이용할 때에는 절반 이상 줄어든 117KB인 것을 확인했다.

디코더 구현에 필요한 메모리 요구량은 하드웨어를 이용할 경우 소프트웨어를 이용할 때보다 절반가량 줄어들지만 실제 메모리 트래픽은 2배 이상 감소하게 된다. 왜냐하면 H.264 디코딩 과정 대부분은 매크로블럭 단위로 처리되는데 하나의 매크로블럭을 처리하기 위해 발생하는 값들을 동일 변수에 반복해서 read/write하기 때문이다. 그렇기 때문에 한 개의 프레임을 처리하더라도, 특정 변수들은 프레임 내의 매크로블럭의 개수만큼 read/write하게 된다. 이 과정을 하드웨어를 이용한 디코더로 처리할 경우 하드웨어 디코더의 내부 메모리와 버스를 사용하기 때문에 MCU 보드의 시스템 버스에 영향을 주지 않아서 메모리 트래픽은 발생하지 않는다.

따라서 메모리 요구량은 하드웨어를 이용할 경우 절반 정도로 감소하지만, 실제 디코딩 동작상의 메모리 트래픽의 양은 그보다 더 감소하게 된다.

4. 소프트웨어와 하드웨어 H.264 디코더 측정 결과 비교

1 절과 2 절에서는 소프트웨어와 하드웨어 H.264 디코더의 성능을 측정해 보았다. 두 가지 경우에서 각 대응되는 모듈별로 비교해보면, 소프트웨어 디코더에서 critical function이 되었던 Intra Prediction과 Motion Compensation 그리고 Deblocking Filter가 하드웨어 디코더의 Codec 부분으로 대체되면서 약 36배까지 성능이 향상된 것을 알 수 있었다. 하지만, Parse 부분에 해당하는 Entropy Decoder의 경우 약 18% 정도의 비교적 미미한 수준의 개선만을 보여주고 있다.

그리고 앞서 설명한 H.264 디코딩 동작 과정을 살펴보면 Codec 부분에 해당하는 모듈들은 데이터 집중적인 동작을 수행하는 데 비해 Parsing 부분에 해당하는 Entropy Decoder는 그렇지 않다는 것을 알 수 있다. 따라서 소프트웨어를 하드웨어로 대체했을 경우 데이터 집중적인 동작들 대부분이 하드웨어 내부적으로 처리되기 때문에 성능 향상이 훨씬 크게 된다. 반면 Entropy Decoder는 상대적으로 MCU의 처리 속도에 의존성이 많은 동작으로 이루어져 있기 때문에 하드웨어로 대체하더라도 성능 차이가 크지 않다는 것을 알 수 있었다.

또 소프트웨어 디코더에서 데이터 집중적인 동작이 bottleneck이 되는 것을 알 수 있었는데, 그 원인으로는 MCU의 처리 속도 문제라기보다는 MCU와 시스템 버스 그리고, 메모리 사이의 속도차이를 들 수 있다. 그림 10은 실험에 사용한 MCU 보드의 MCU와 시스템 버스 및 메모리의 속도차이를 보여준다. 이와 같은 시스템에서는 H.264 디코더 같은 데이터 집중적인 동작이 그렇지 않은 애플리케이션보다 MCU와 다른 장치

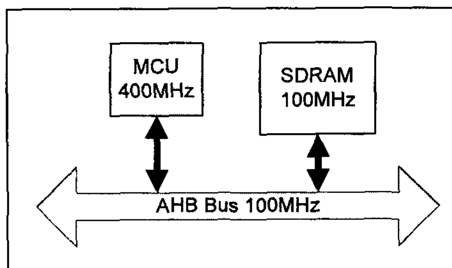

그림 10. MCU 보드의 MCU와 Bus 및 메모리 구조
Fig. 10. Internal organization of MCU board.

간의 속도 차이가 성능에 큰 영향을 줄 수 있다는 것을 알 수 있다.

Ⅶ. 펌웨어 환경에서의 성능 측정 결과

표 6은 펌웨어 환경에서 단일 프로세스만 동작하는 하드웨어 H.264 디코더의 time complexity를 측정한 결과이다. O/S의 유무를 제외하고는 기본적인 플랫폼은 ARM linux 환경과 동일하다. 결과를 보면 ARM linux 환경에서 동일 실험을 한 것보다 time complexity가 낮아진 것을 확인할 수 있다. 이 차이는 O/S의 경우 시스템을 유지하기 위해 기본적인 프로세스들이 동작하고 있기 때문에 그것으로 인한 오버헤드가 존재하기 때문이다. 이런 종류의 프로세스들은 다른 애플리케이션들이 동작하는 동안에서 실행되고 있어서 펌웨어 환경에서는 발생하지 않았던 오버헤드가 된다.

표 7은 ARM linux 환경에서 측정한 결과와 펌웨어 환경에서 측정한 결과의 차이를 표로 나타낸 것이다. 이 차이는 O/S로 인해서 발생하는 오버헤드로 하드웨어 H.264 디코더를 동작할 경우를 기준으로 한 프레임당 277.410 μsec의 시간이 더 걸린 것을 알 수 있다. 25fps로 인코딩된 H.264 비트스트림을 처리한다고 할 때, 초당 6935.26 μsec가 순수 디코딩 작업과 관계없이 발생하는 오버헤드가 된다. 이런 차이 때문에 실제 임베디드 O/S 환경에서 멀티미디어 시스템을 구현할 때

표 6. 펌웨어에서 측정한 하드웨어 H.264 디코더의 Time Complexity
Table 6. Time complexity of hardware H.264 decoder with firmware driver.

Modules	Time (μsec)	Clock Cycle
Parsing	923.020	3692.080
Codec	4296.045	17184.180
Host I/F Read	12908.070	51632.280
Host I/F Write	249.894	999.578
TOTAL	18377.029	73508.118

표 7. H.264/AVC 디코더에서 O/S로 인한 오버헤드
Table 7. O/S overhead in hardware H.264/AVC decoder.

Modules	Time (μsec)	Clock Cycle
Parsing	87.320	349.279
Codec	22.085	88.340
Host I/F Read	37.920	151.680
Host I/F Write	130.085	520.342
Total	277.410	1109.641

에는 이렇게 발생하는 오버헤드를 감안해서 멀티미디어 디코더 등의 성능을 테스트해야 한다.

Ⅷ. 결 론

이 논문에서는 H.264 디코더의 성능을 임베디드 환경에서 실측하는 방법을 제안하였다. 제안한 방법은 H.264 디코더의 성능을 측정함에 있어 디코더의 동작에 오버헤드를 최소화 하면서 정확도를 높이는 데 효과적이다. 이 방법을 이용해서 H.264 디코더를 ARM linux 환경과 펌웨어 환경에서 측정하여 O/S 기반 환경에서 O/S의 백그라운드 프로세스로 인한 오버헤드를 구할 수 있었다. 이런 결과들을 H.264 재생기를 구현할 때, 구현 목적에 따라 플랫폼을 구성에 참고할 수 있을 것이다.

그 외에도 디코더의 실측 결과를 보면 데이터 집중적인 동작을 주로 수행하는 Intra Prediction, Motion Compensation, Deblocking Filter 등이 H.264 디코딩 과정에서 critical function이 되는 것을 알 수 있었는데, 이상의 동작은 하드웨어로 변경했을 경우 높은 성능 향상을 보였지만, 그 외의 Entropy Decoder 같은 동작의 경우 하드웨어도 변경하더라도 미미한 수준의 성능 변화만이 나타났다. 이런 결과를 가지고, 임베디드 환경에서 H.264 디코딩을 할 때 과부하가 발생하는 원인은 데이터 집중적인 동작을 수행하면서 발생하는 데이터 트래픽이 원인인 것을 알 수 있었다.

참 고 문 헌

[1] Michael Horowitz, Anthony Joch, and Faouzi Kossentini, "H.264/AVC Baseline Profile Decoder Complexity Analysis," IEEE Transactions on Circuits System, 13(7), pp. 704-716, 2003
[2] Abu Asaduzzaman, and Imad Mahgoub, "Cache Optimization for Embedded Systems Running MPEG-4 Video Decoder," Multimedia Tools Application, Vol. 28, No. 1-2, pp. 239-256, 2006
[3] Youngsoo Kim, and Suleyman Sair, "Designing Real-Time H.264 Decoders with Dataflow Architecture," CODES+ISSS, pp.291-296, ACM, 2005
[4] Tung-Chien Chen, Chung-Jr Lian, and Liang-Gee Chen, "Hardware Architecture Design of an H.264/AVC Video Codec," ASP-DAC, pp.750-757, IEEE, 2006
[5] Iole Moccagatta, "Recent Developments in Video Compression Standards and their Impact on Embedded Platforms: from Scalable to Multi-view Video Coding," ACM Multimedia, p.11, ACM, 2006
[6] Jian-Wen Chen, Chao-Yang Kao and Youn-Long Lin, "Introduction to H.264 Advanced Video Coding," IEEE, 2006
[7] Aleksandar Milenkovic, Milena Milenkovic, and Nelson Barnes, "A Performance Evaluation of Memory Hierarchy in Embedded Systems," IEEE, 2003
[8] Andrew N. Sloss, Dominic Symes, Chris Wright, "ARM System Developer's Guide: Designing and Optimizing System Software," Elsevier, 2004
[9] "ARM920T Technical Reference Manual," ARM, 2001
[10] Sung Dae Kim, Jeong Hoo Lee, Chung Jin Hyun, Myung Hoon Sunwoo, "ASIP Approach for Implementation of H.264/AVC," ASP-DAC, pp. 758-764, IEEE, 2006
[11] "S3CA480X01 User's Manual," Samsung, 2004. "Information technology - Coding of audio-visual objects - Part10: Advanced Video Coding," ISO/IEC, 2004
[12] "S3C2440A 32-Bit RISC Microprocessor User's Manual," Samsung, 2004
[13] Iain E. G Richardson, "H.264 and MPEG-4," 2004
[14] Tsu-Ming Liu, Ching-Che Chung, Chen-Yi Lee, Ting-An Lin, Sheng-Zen Wang, "Design of a 125μW, Fully-Scalable MPEG-2 and H.264/AVC Video Decoder for Mobile Applications," DAC, pp. 288-289, ACM, 2006
[15] Shin-Haeng Ji, Jung-Wook Park, and Shin-Dug Kim, "Optimization of Memory Management for H.264/AVC Decoder," ICACT, 2006
[16] Krishna V. Palem, Rodric M. Rabbah, Vicent J. Mooney III, Pinar Korkmaz, Kiran Puttaswamy, "Space Optimization of Embedded Memory Systems via Data Remapping," ACM 2003
[17] Arijit Ghosh, Tony Givargis, "Cache Optimization For Embedded Processor Cores: An Analytical Approach", ICCAD, 2003
[18] VideoLAN-VLC media player website, http://videolan.org
[19] FFMPEG Multimedia System website, http://ffmpeg.mplayerhq.hu
[20] Y. Chen, E. Li, X. Zhou and S. Ge, "Implementation of H.264 encoder and decoder on personal computers," Journal of Visual Communications and Image Representation, No. 2, Vol. 17, pp. 509-532, 2006.

저 자 소 개

정 진 원(학생회원)
2003년 광운대학교 컴퓨터공학과, 학사
2007년 한양대학교 전자컴퓨터통신공학과, 석사
<주관심분야 : 임베디드 시스템, SoC>

송 용 호(정회원)
1989년 서울대학교 컴퓨터공학과, 학사.
1991년 서울대학교 컴퓨터공학과, 석사.
2002년 University of Southern California, Electrical Engineering, Ph.D.
<주관심분야 : 임베디드시스템, SoC, NoC>