논문 2009-46SD-8-14

멀티코어 프로세서에서의 H.264/AVC 디코더를 위한 데이터 레벨 병렬화 성능 예측 및 분석

(Data Level Parallelism for H.264/AVC Decoder on a Multi-Core Processor and Performance Analysis)

조 한 욱*, 조 송 현*, 송 용 호**

(Han Wook Cho, Song Hyun Jo, and Yong Ho Song)

요 약

최근 멀티코어 프로세서의 이용이 증가함에 따라, 멀티코어환경에서 고성능 H.264/AVC 코덱을 구현하기 위한 다양한 병렬화 기법들이 제안되고 있다. 이러한 기법들은 병렬화 기법 적용 방식에 따라 태스크 레벨 병렬화 기법과 데이터 레벨 병렬화 기법으로 구분된다. 태스크 레벨 병렬화 기법을 이용한 파이프라인 병렬화 기법은 H.264 알고리즘을 파이프라인 단계로 나누어 구현하며, 일반적으로 화면 사이즈가 작고 복잡도가 낮은 비트스트림에 유리하다. 그러나 프로세싱 모듈별 수행시간 차이가 커서 로드밸런싱이 좋지 않고, 파이프라인 단계의 수가 제한적이라 성능 확장성에 제한이 있어 HD 비디오같이 해상도가 큰 비트스트림 처리에는 적합하지 않은 단점이 있다. 본 논문에서는 로드밸런싱 및 성능 확장성을 고려하여 매크로블록 라인 단위로 쓰레드를 할당하는 수평적 데이터 레벨 병렬화 기법을 제안하고, 이에 대한 성능 예측 수식 모델을 통하여 성능을 예상한다. 또한 성능 예측의 정확성을 검증하기 위해 JM 13.2 레퍼런스 디코더에 대한 데이터 레벨 병렬화 기법을 ARM11 MPCore 환경에서 구현하고 이에 대한 성능 검증을 수행하였다. SoCDesigner를 이용한 사이클 단위의 성능 측정 결과, 본 논문에서 제시하는 쓰레드 증가에 대한 병렬화 기법의 성능 변화를 비교적 높은 수준의 정확도로 예측 가능하였다.

Abstract

There have been lots of researches for H.264/AVC performance enhancement on a multi-core processor. The enhancement has been performed through parallelization methods. Parallelization methods can be classified into a task-level parallelization method and a data level parallelization method. A task-level parallelization method for H.264/AVC decoder is implemented by dividing H.264/AVC decoder algorithms into pipeline stages. However, it is not suitable for complex and large bitstreams due to poor load-balancing. Considering load-balancing and performance scalability, we propose a horizontal data level parallelization method for H.264/AVC decoder in such a way that threads are assigned to macroblock lines. We develop a mathematical performance expectation model for the proposed parallelization methods. For evaluation of the mathematical performance expectation, we measured the performance with JM 13.2 reference software on ARM11 MPCore Evaluation Board. The cycle-accurate measurement with SoCDesigner Co-verification Environment showed that expected performance and performance scalability of the proposed parallelization method was accurate in relatively high level

Keywords: H.264, Multi-Core, Thread Level Parallelism, Data Level Parallelism, OpenMP

I. 서 론

최근 네트워크 및 멀티미디어 기술 발달로 고성능 연산이 필요한 애플리케이션이 증가함에 따라 프로세서 성능에 대한 요구가 증가하고 있다. 기존에는 프로세서의 클럭 주파수 향상을 통한 성능 향상 방법을 적용하여 왔으나, 반도체 공정상의 제한, 고열 및 높은 전력 소모 등의 문제로 한계점에 부딪히고 있었다. 여러 개의 프로세서 코어를 하나의 칩에 집적하는 멀티코어 프

* 학생회원, ** 정회원, 한양대학교
(Hanyang University)
접수일자: 2009년4월8일, 수정완료일: 2009년7월31일

로세서는 병렬처리가 가능하므로 상대적으로 낮은 클럭 주파수로 고성능의 시스템 구현이 가능하다. 또한 높은 클럭 주파수가 필요 없으므로 저 전력 시스템 구성이 가능하여 인텔의 쿼드 코어 프로세서같은 서버용 프로세서에서부터 ARM11 MPCore같은 임베디드용 프로세서에 이르기까지 멀티코어구조가 광범위하게 사용되고 있다.

Mobile TV, IPTV 등 차세대 멀티미디어 디지털 방송과 화상회의시스템, 캠코더, 멀티미디어 메세징, 멀티미디어 플레이어, 게임 등 멀티미디어 응용 소프트웨어에서 많이 사용되고 있는 H.264/AVC 코덱에 대해 성능 향상을 위한 다양한 연구들이 있어 왔다. H.264/AVC는 높은 압축률 및 고화질 지원을 위해 MPEG-2, MPEG-4 등 기존 비디오 코덱보다 복잡한 알고리즘을 사용하기 때문에 고성능 프로세서를 필요로 한다. 하지만 멀티코어 프로세서를 위한 고성능의 H.264/AVC 구현은, 여러 코어가 동시에 동작될 수 있도록 그림 1과 같이 H.264/AVC 코덱의 효율적인 병렬화 (Parallelization)가 관건이다.

멀티코어의 시스템 레벨에서의 H.264/AVC 디코더 병렬화 기법은, 프로세싱 모듈을 태스크로 할당하는 파이프라인 형태의 태스크 레벨 병렬화 기법과 H.264/AVC 데이터를 병렬화가 가능하게 나누어 처리하는 데이터 레벨 병렬화 기법으로 나눌 수 있다. 프로세싱 모듈의 수행 시간 차이 때문에 로드밸런싱(Load Balancing)이 좋지 않아 태스크 레벨 병렬화 기법의 성능 향상에 어려움이 있고, 쓰레드의 개수가 파이프라인 단계 수에 제한이 되기 때문에 성능 확장성이 좋지 못하여 복잡하고 데이터가 큰 고화질 비디오 처리에 단점이 있다.

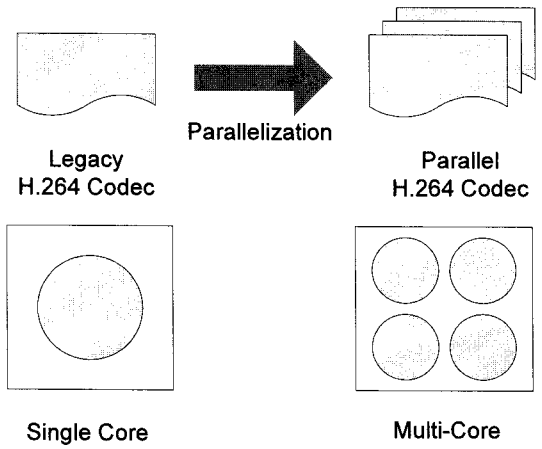


그림 1. H.264/AVC Codec 병렬화
Fig. 1. H.264/AVC Codec Parallelization.

데이터 레벨 병렬화 기법은 태스크 레벨 병렬화 기법보다 성능 향상 및 성능 확장성이 좋으나, 데이터 의존 및 메모리 오버헤드가 최소화되도록 데이터의 분할 기준에 따라 성능 및 성능 확장성이 달라진다. 따라서 고성능의 데이터 레벨 병렬화 기법에서는 데이터 분할 단위 선택과 성능 확장성 및 데이터 의존도를 고려한 쓰레드 스케줄링이 중요하다. 기존의 데이터 레벨 병렬화 기법들은 데이터 의존 및 H.264/AVC 디코더 프로세싱 모듈의 동작 흐름 분석이 부족하여 H.264/AVC 표준에 맞지 않는 문제점이 있다.

본 논문은 H.264/AVC 표준에 부합하기 위해 H.264/AVC 프로세싱 모듈 중 Deblocking Filter를 분리하여 슬라이스 단위로 처리하고, 성능 확장성 및 데이터 의존도를 고려하여 쓰레드를 매크로블록 라인에 할당하는 방식의 수평적 데이터 레벨 병렬화 기법을 제안한다. 또한 제안된 기법에 대한 수식적 성능 분석 및 성능 예측이 실제와 비교적 높은 수준의 정확성을 보임을 실험 결과로 보여준다.

Ⅱ장에서 H.264/AVC 병렬화 기법에 대한 제반 기술 및 관련 연구들을 살펴보고, Ⅲ장에서 제안된 H.264/AVC 디코더 수평적 데이터 병렬화 기법을 설명하고 수식적으로 성능을 분석 및 예측한다. Ⅳ장에서 예측 성능과 실험 결과를 비교 분석한 후, Ⅴ장에 결론으로 마무리한다.

Ⅱ. 배경 및 관련 연구

1. H.264/AVC 디코더 개요

H.264 혹은 MPEG-4 AVC는 ISO/IEC 와 ITU가 함께 만든 비디오 압축 표준으로 MPEG-2, MPEG-4 등 기존에 많이 쓰이는 비디오 압축 표준보다 더 높은 압축율 가지며, 네트워크를 통한 스트리밍에 적합한 특성을 가지고 있어서 다양한 멀티미디어 응용 분야에 많이 쓰이고 있다. 본 절에서 H.264/AVC의 구조에 대해서 간략히 설명한다. H.264/AVC 표준 및 사용되는 알고리즘에 대한 자세한 사항은 ITU의 H.264 표준[1], ISO MPEG-4/AVC 표준[2]혹은 표준의 개요를 설명한 문서[3]를 참조하기 바란다.

가. H.264/AVC 데이터 단위 구조

H.264/AVC는 그림 2에서 도시된 바와 같이

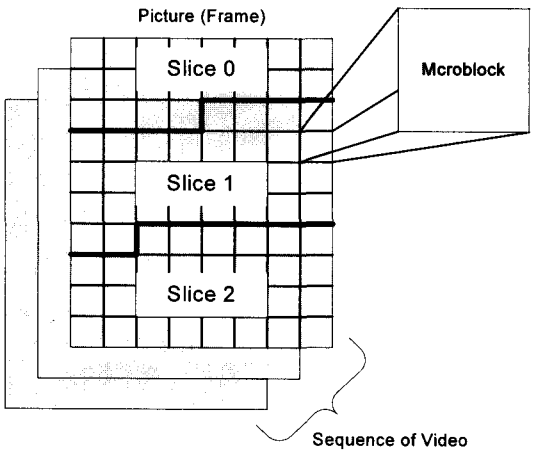


그림 2. H.264/AVC 데이터 단위
Fig. 2. H.264/AVC Data Units.

MPEG-2 또는 MPEG-4에서 사용하는 데이터 구조와 유사한 구조를 사용한다. 즉, 비디오 시퀀스 (Sequence of Video)는 여러 개의 GOP(Group Of Picture)로 구성되며, GOP는 다시 여러 개의 픽쳐 (Picture) 혹은 프레임(Frame)으로 구성된다. 또한 하나의 픽쳐는 하나 이상의 슬라이스(Slice)로 나뉜다. 슬라이스는 여러 개의 매크로블록(Macroblock)으로 구성되며, 매크로블록은 luminance 블록 및 chroma 블록으로 구성된다. 슬라이스들은 다른 슬라이스와 독립적으로 코딩된다. 슬라이스에는 공간적 중복성(spatial redundancy)을 이용하여 코딩되는 I-슬라이스와 시간적 중복성(temporal redundancy)을 이용하는 P-슬라이스와 B-슬라이스 등 세 가지 타입이 있다.

매크로블록은 더 작은 크기의 8x8의 서브블록으로 나누어진다. 매크로블록 헤더에 있는 CBP(Coded Block Pattern) 값에 따라서 서브블록의 코딩여부를 판단한다. CBP 의 해당 비트가 0인 서브블록은 코딩되어 있지 않아, 엔트로피 디코딩, 역변환(inverse transform and inverse quantization) 같은 디코딩 작업이 필요 없다. 다음과 같이 세 가지 종류의 매크로블록이 있다.

- *Intra-coded 매크로블록*: 공간적 중복성만을 이용하여 움직임예측이 없는 것을 Intra-coded 매크로 블록이라 한다. I-슬라이스는 이러한 타입의 매크로블록으로만 구성된다.
- *Inter-coded 매크로블록*: 시간적 중복성을 이용하여 코딩되는 블록을 Inter-coded 매크로 블록이라 한다.
- *Skipped 매크로블록*: 참조 프레임의 해당 매크로블록과 차이가 적은 매크로블록을 코딩한 것으로, 복원시 참조 프레임의 블록을 그대로 사용한다. 따라서 엔트로피 디코딩과 역변환이 필요 없다. 이러한 매크로블록을 Skipped 매크로블록이라 한다.

나. H.264/AVC 디코더 주요 동작

H.264/AVC 디코더는 Entropy Decoding, Inverse Transformation 및 Inverse Quantization, Inter-Frame Prediction, Intra-Frame Prediction, 그리고 In-Loop Deblocking Filter로 구성되며 동작 흐름도는 그림 3과 같다.

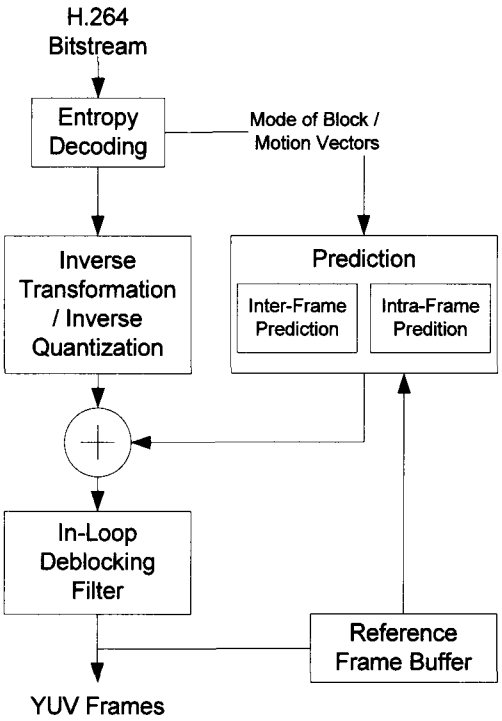


그림 3. H.264/AVC 디코더 블록 다이어그램
Fig. 3. H.264/AVC Decoder Block Diagram.

(1) ED(Entropy Decoding)

H.264/AVC 비트스트림은 NAL(Network Adaptation Layer) 단위의 패킷형태로 입력되어, 그림 3에서와 같이 ED프로세싱 모듈 작업이 적용된다. ED 작업을 거치면 다음 프로세싱 모듈 IDCT의 입력인 계수(Coefficient)들이 생성된다. H.264/AVC에서는 문맥(Context)에 따라 효율적인 엔트로피 코딩이 가능한 형태의 CAVLC (Context-Adaptive Variable Length Coding)와 CABAC (Context-Adaptive Binary Arithmetic Coding) 두 가지의 엔트로피 디코딩을 지원한다. 베이스라인 프로파일(Baseline Profile)에는

CAVLC가 사용되고, 메인 프로파일(Main Profile), 하이프로파일(High Profile)등에는 CABAC이 사용된다.

(2) IDCT(Inverse Transformation), IQ(Inverse Quantization)

엔트로피 디코딩 작업 후 생성되는 계수들은 IDCT와 IQ과정을 거친다. 기존 비디오 코덱과 같이 DCT (Discrete Cosine Transformation) 알고리즘을 사용하나, 정수만을 사용하는 변환 알고리즘을 사용하므로, IDCT시 발생하는 불일치(mismatch) 문제가 없다. 기존 코덱과 같이 기본적으로는 4x4 블록에 대해서 DCT가 적용되나, 부드러운 영역을 위한 코딩을 위해서 4x4 블록 그룹의 DC 계수에 대해서 한 번 더 DCT를 수행하여 코딩된다.

(3) PRED(Prediction)

Prediction은 매크로블록의 타입에 따라서 Intra-Frame prediction을 적용하거나, Inter-Frame prediction을 적용한다. 참조 프레임이 필요 없는 Intra-coded 매크로블록에는 Intra-Frame prediction을, Inter-coded 매크로블록에는 Inter-Frame prediction이 적용된다. prediction후 만들어지는 블록은 IDCT, IQ 후에 만들어지는 residual 블록과 합쳐진다.

공간적 중복성을 이용하는 Intra-Frame prediction은, Intra-coded 매크로블록에 적용되며, Intra_4x4 prediction 모드와 Intra_16x16 모드 두 가지가 지원된다. Intra_coded 4x4 모드에서는 수평(Horizontal), 수직(Vertical), 대각선(Diagonal) 등 prediction방향에 따른 8가지 모드와 DC prediction을 포함한 9가지 모드를 지원한다. Intra_16x16 모드에서는 DC, 수평, 수직, plane 네 가지 모드의 prediction을 지원한다.

시간적 중복성을 이용하는 Inter-Frame Prediction 혹은 Motion Compensation은 Inter-coded 매크로블록에 적용된다. 움직임 예측을 위한 움직임 벡터(Motion Vector)는 기존 MPEG-2나 MPEG-4와는 달리 16x16, 16x8, 8x8, 8x4, 4x4등 다양한 블록사이즈를 이용하여 만들어 지므로, 기존 비디오 코덱보다 정밀한 움직임을 추출할 수 있고, 하나이상의 참조 프레임을 사용하므로 더 정확한 움직임 예측이 가능하다.

(4) DF(In-Loop Deblocking Filter)

대부분의 비디오 압축 코덱은 블록으로 나누어 코딩하는 방식을 취한다. 디코딩된 이미지들은 블록간의 경계가 뚜렷이 나타나는 블록킹(blocking)이 나타난다. 이를 없애기 위해서 블록경계를 스무딩(smoothing) 하는 Deblocking Filter를 적용한다. H.264/AVC에서는 스무딩시 사용되는 가중치 값을 제어하면서 적용하는 적응형(adaptive) 방법을 사용하여 정확한 그림을 복원할 수 있도록 한다.

2. H.264/AVC 디코더 분석

H.264/AVC 비디오는 소수의 I 프레임과 다수의 P 프레임 혹은 B 프레임으로 구성된다. I 프레임은 주로 인덱스 탐색 혹은 급격한 장면 전환이 있는 프레임에 이용되어 전체적으로 볼 때 많은 비중을 차지하지 않는다. 또한 P 프레임을 디코딩하기 위해서는 ED, IDCT와 IQ, PRED, DF등의 모든 프로세싱 모듈을 필요로 하므로 P 프레임 디코딩은 멀티코어 성능분석에 충분한 정보를 제공한다. 본 논문에서는 QCIF 크기의 P 프레임을 디코딩하는 것으로 분석하고, 편의상 IDCT 프로세싱 모듈이 IDCT와 IQ를 포함한다.

디코더 분석을 위하여, H.264/AVC Baseline Profile로 인코딩된 QCIF 크기의 대표적인 두 가지 비트스트림을 이용하였다. 하나는 움직임이 다소 적은 Akiyo이고, 다른 하나는 Akiyo보다 움직임이 다소 많은 Foreman이다. 그림 4는 Akiyo P 프레임 매크로블록 패턴과 Foreman P 프레임 매크로블록 패턴이다. 움직임이 적은 H.264/AVC 비트스트림은 많은 Skipped 매크로블록으로 구성된 반면 움직임이 많은 비트스트림은 Inter-Coded 매크로블록이 많다.

이들 비트스트림들을 H.264/AVC 공식 참조 소프트웨어인 JM 13.2[12]을 이용하여 디코딩하여 측정한 결과가 그림 5와 그림 6이고, 표 1은 측정된 각 모듈 수행시간의 상대적인 관계를 보기 위해서 ED 수행시간을 1

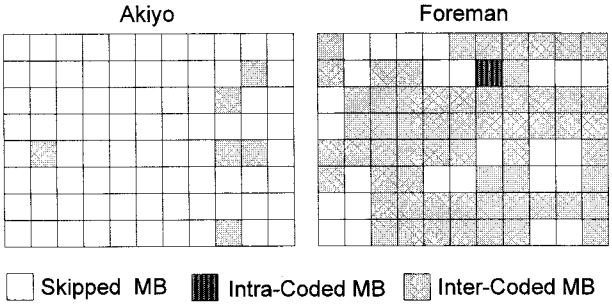
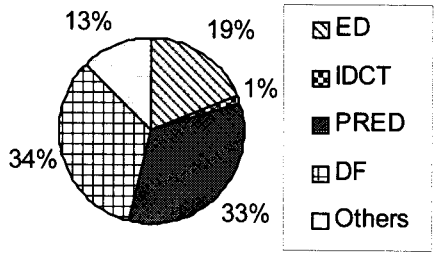
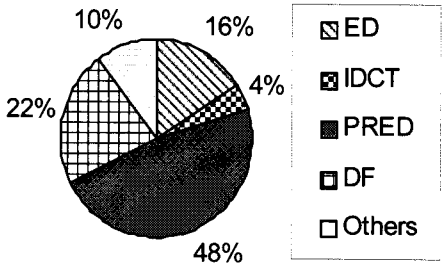


그림 4. P 프레임 매크로블록 타입 패턴
Fig. 4. Macroblock Type Patterns of a P Frame.



(a) Akiyo



(b) Foreman

그림 5. H.264/AVC 비디오 디코더의 P 프레임 성능 프로파일

Fig. 5. Performance Profile of H.264/AVC P Frame Decoding.

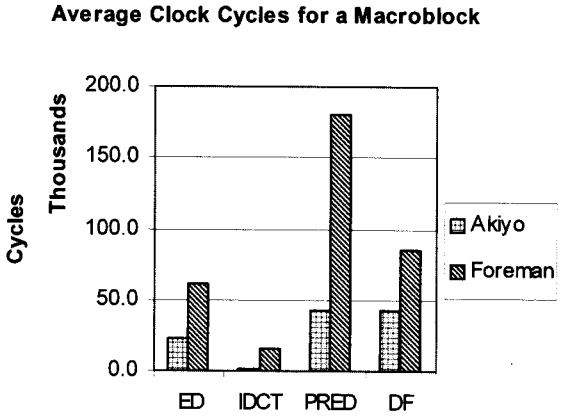


그림 6. 한 매크로블록 처리시의 각 프로세싱 단계별 평균 클럭 사이클수

Fig. 6. Average Clock Cycles of Each Processing Modules When Decoding One Macroblock.

로 했을 때의 상대시간을 나타낸 것이다. 그림 5의 성능 프로파일에서 보듯이, Skipped 매크로블록이 많은 비중을 차지하는 Akiyo의 경우, 코딩된 데이터가 적으므로 ED와 IDCT가 Foreman에 비해 적은 비중을 차지함을 볼 수 있다. Foreman의 경우도 P 프레임에서는 움직임 혹은 색등의 변화 정도가 그리 크지 않아서

표 1. 프로세싱 단계별 표준 수행시간

Table 1. Normalized Processing Stage Execution Time.

Processing Stage	Akiyo	Foreman
ED	1.00	1.00
IDCT	0.07	0.27
PRED	1.81	2.96
DF	1.81	1.40

Intra-coded 블록이 매우 적으며, 대부분의 Inter-coded 블록등도 CBP에서 설정된 코딩된 서브블록이 적어 상대적으로 IDCT의 비중이 작음을 볼 수 있다.

Akiyo와 Foreman에서 하나의 매크로블록을 디코딩할 때 프로세싱 모듈의 수행 시간을 측정하여 비교한 것이 그림 6이다. 그림 6을 보면 Akiyo가 처리 시간이 Foreman에 비해서 상당히 작다. 특히 Skipped 매크로블록이 많아서 PRED 수행시간이 Foreman에 비해서 상당히 작다. 표 1에서도 그림 5와 그림 6과 같이 PRED와 DF의 수행시간이 길며, 특히 Foreman에서는 PRED가 상대적으로 크다.

3. 멀티코어 아키텍쳐와 OpenMP

본 논문에서는 멀티코어 구조를 그림 7과 같이 Shared memory 형태의 SMP(Symmetric Multi-Processor) 구조를 가정한다. 이러한 구조는 Intel및 AMD의 Quad-Core CPU와 ARM11 MPCore[4] 등 많은 프로세서에서 볼 수 있다.

OpenMP는 Shared memory 기반의 멀티코어 혹은 멀티프로세서를 위한 병렬 프로그래밍을 위한 API (Application Programming Interface)이다[5]. C/C++, FORTRAN 등의 directive형태로 제공하기 때문에 사용하기 쉬워 최근 많이 이용되고 있다. 본 논문에서는 H.264/AVC의 병렬성 분석을 위해 여러 개의 쓰레드(thread)가 동시에 동작하는 multi-threaded 프로그래밍

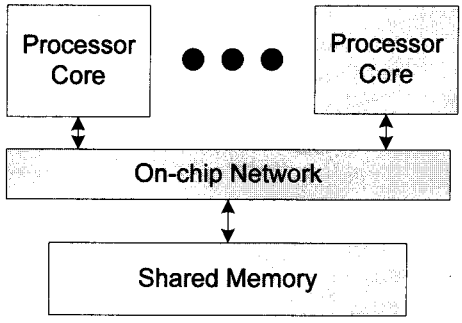


그림 7. 공유메모리 멀티코어 아키텍쳐

Fig. 7. Shared Memory Multi-core Architecture.

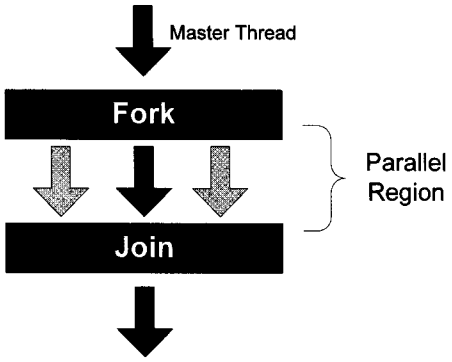


그림 8. OpenMP Fork-Join 모델
Fig. 8. OpenMP Fork-Join Model.

이 필요한데, OpenMP는 사용하기 쉽고 간단한 병렬 프로그래밍 환경을 제공한다. OpenMP는 병렬 수행이 가능한 모듈을 Fork하여 동시 수행하도록 하고, Join을 통해서 모듈 수행의 끝을 맞춘다. 이것을 Fork-Join 모델이라 한다. 본 논문은 OpenMP의 Fork-Join 모델을 이용하여 H.264/AVC 병렬성 분석 및 실험을 진행했다.

4. 관련 연구

고성능 비디오 디코더 구현 연구들은, MPEG-2, MPEG-4, H.264/AVC 등 비디오 디코더에서 많은 계산이 필요한 프로세싱 모듈을 명령어 레벨 병렬화 분석을 통해 프로세서가 제공하는 SIMD(Single Instruction Multiple Data) 명령어를 이용하여 최적화하는 것[6~7]과 병렬 구조를 분석한 병렬 시스템 레벨 병렬화를 통한 성능향상연구로 나눌 수 있다[8~12].

SIMD를 이용한 최적화는 특정 알고리즘 블록을 DSP(Digital Signal Processor) 같이 SIMD가 가능한 프로세서에서 최적의 성능을 보이도록 구현하는 것으로 시스템 레벨의 병렬화를 이용하는 것은 아니다. 그러나 프로세서 코어가 인텔의 MMX 처럼 SIMD extension을 가지고 있는 경우에는 멀티코어의 프로세싱 모듈 최적화에 이용 될 수 있다.

멀티코어 병렬화 구조에 초점을 맞춘 병렬화 기법은 동시 수행이 가능한 태스크로 나누어서 여러 태스크를 동시에 수행하게 하는 태스크 레벨 병렬화 (TLP: Task Level Parallelization) 기법과 전체 데이터를 잘 나누어 여러 프로세서가 동시에 처리하게 하는 데이터 레벨 병렬화(DLP: Data Level Parallelization) 기법으로 나눌 수 있다.

가. 태스크 레벨 병렬화 기법 (TLP: Task Level Parallelization)

H.264/AVC 디코더는 그림 9와 같이 H.264/AVC 비트스트림을 ED, IDCT, PRED, DF 프로세싱 모듈에서 순차적으로 처리된 후, YUV 비디오 출력을 만들어 낸다. 그림 9에서와 같이 각 프로세싱 모듈에 해당하는 태스크를 쓰레드(Thread)에 할당하여 처리 단계들이 동시에 처리될 수 있도록 병렬화한다. H.264/AVC의 프로세싱 모듈의 기능 및 입력/출력 형태가 명확하여 각 프로세싱 모듈을 하드웨어 모듈로 구현하기가 용이하며, 보통 그림 10에서와 같이 4단계 파이프라인 형태로 처리한다. 매크로블록단위로 입력된 H.264/AVC 비디오는 매크로블록단위로 각 단계별 처리를 거쳐서 매크로 블록단위로 출력된다.

그림 11은 OpenMP의 Fork-Join 모델로 구현했을 때, 태스크 스케줄링의 예를 나타낸 것이다. 빗금친 영역은 코어가 아무 일도 하지 않는 idle한 상태를 나타낸다. 전체 수행시간은 파이프라인 단계에서 가장 긴 수행시간들을 합한 것과 같다. 이러한 구조에서는 모듈간의 수행시간의 차이가 크면, 수행시간이 작은 모듈을 수행하는 코어의 idle한 상태가 프로세싱 모듈사이의 수

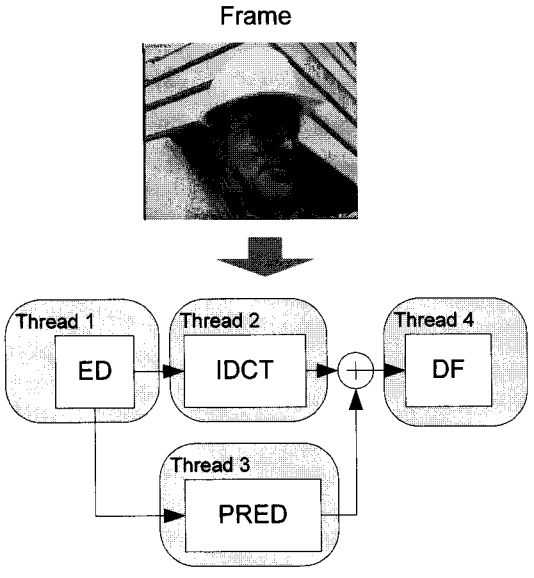


그림 9. H.264/AVC 디코더 태스크레벨 병렬화
Fig. 9. H.264/AVC Decoder Task Level Parallelization.

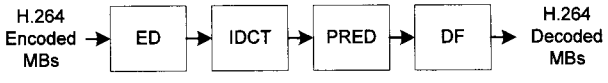


그림 10. 4단계 파이프라인 H.264/AVC 디코더
Fig. 10. 4 Satge Pipeline for H.264/AVC Decoder.

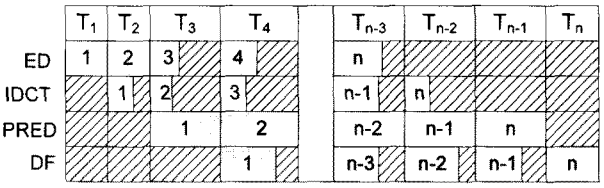


그림 11. 파이프라인 병렬화에서 쓰레드 스케줄링
Fig. 11. Thread Scheduling in Pipelined Parallelization.

행 시간 차이만큼 크게 된다.

병렬화 기법이 최상의 성능을 보이기 위해서는 로드 밸런싱이 잘 되어야 한다. 즉, 각 프로세서 코어의 idle 상태 시간이 작아야 한다. 이를 위해서는 각 프로세싱 모듈의 수행 시간의 차가 작아야 한다. 그러나 표 1에서와 같이 H.264/AVC 디코더의 각 프로세싱 모듈간의 수행 시간 차이가 크다. P 프레임의 경우, PRED에서 가장 많은 시간이 소요되며, IDCT에서 가장 적은 시간을 필요로 한다. 같은 IDCT라 하더라도, 변화가 큰 매크로블록이 변화가 작은 매크로블록보다 더 많은 수행 시간을 필요로 한다. 2절의 Akiyo와 Foreman의 프로파일 결과를 통해 알 수 있는 것처럼 모듈별 수행 시간이 비트스트림마다 다르다. 또한 동일한 비트스트림내에서도 프레임마다 복잡도가 다르기 때문에 프레임마다 그 수행 시간이 다르다. 따라서 모듈사이의 수행 시간 변화가 커서 프로세싱 모듈단위로 할당되는 쓰레드의 로드밸런싱이 좋지 못하다.

HD 비디오와 같이 복잡하고 데이터 크기가 큰 비트스트림의 경우, 모듈별 데이터 이동에 필요한 오버헤드로 인하여 코어 간 데이터 통신 대역폭이 충분하지 않는 한 충분한 성능 향상을 기대하기 어렵고 성능 확장성(Scalability)에 있어서 문제가 있다. H.264/AVC는 일반적으로 4개의 파이프라인 단계로 나누어지는데, 코어의 개수가 이보다 많더라도, 각 모듈들이 입력/출력 관계로 연결되는 sequential한 특성으로 인해 성능향상에 별 도움을 주지 못한다.

그러나 복잡도가 그리 높지 않고, 프레임 크기가 작은 경우 표 1의 Akiyo 처럼 프로세싱 모듈간의 수행 시간 차이가 크지 않아 적은 개수의 코어나 하드웨어 블록으로 각 모듈을 구현하는 데 많이 이용될 수 있다.

나. 데이터 레벨 병렬화 기법 (DLP: Data Level Parallelization)

H.264/AVC 디코더를 단계별로 나누어 태스크로 하는 대신에, 그림 12와 같이 모든 단계를 포함하는 디코더 전체를 하나의 쓰레드로 할당한다. 데이터는 여러 개로 나누어 각 태스크, 즉 비디오 디코더가 독립적으로 디코딩 할 수 있도록 나눈다. 이때, 각 데이터의 지역성(locality)로 인하여 태스크간의 데이터 통신이 TLP에 비해 적다. 따라서 디코딩해야 할 데이터가 크더라도 태스크 레벨 병렬화 기법처럼 코어 간 데이터 통신을 위한 오버헤드가 작다. 이때 데이터 의존성을 고려하여 데이터를 어떻게 나누어 쓰레드에 할당하는가가 중요한 요소이다.

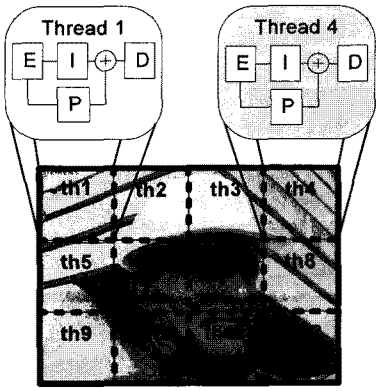


그림 12. H.264/AVC 디코더 데이터 레벨 병렬화
Fig. 12. H.264/AVC Decoder Data Level Parallelization.

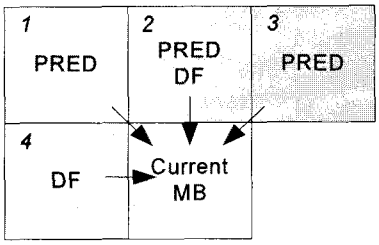


그림 13. 매크로블록 데이터 의존성
Fig. 13. Data Dependencies for a Macroblock.

H.264/AVC 인코더에 대해서 이와 같은 기법을 적용한 연구[8]에서 좋은 성능 향상을 보였으며, DLP를 H.264/AVC 비디오 디코더에 적용한 사례들[10~12]이 있다. 그러나 매크로블록 레벨에서 적용한 DLP 관련 대부분의 연구들에서 H.264/AVC 표준 정합성에 문제가 있다. 즉, 어떤 매크로블록에 대해 Intra Frame Prediction이 수행하기 위해 그 주변 매크로블록을 참조할 때는, 그 주변 매크로블록이 Deblocking Filter가 수행되지 않은 상태이어야 한다. 기존의 병렬화 기법에서는 그림 13에서 1, 2, 3, 4 블록들은 모든 디코딩 작업이 끝난 매크로블록들이다. Current MB가 Intra-coded 매크로블록일 경우, 1, 2, 3에 있는 픽셀의 값이 Prediction에 이용된다. H.264/AVC 표준에 따르면, 이 픽셀은

Deblocking Filter가 적용되기 이전의 값이어야 한다. 그러나 이미 작업이 끝난 블록들이라 Deblocking Filter 적용후의 값으로 Prediction을 하게 되어 H.264/AVC 표준에 위배된다.

III. H.264/AVC 디코더 데이터 레벨 병렬성

1. 데이터 단위 및 매크로블록 데이터 의존성

TLP 기법인 파이프라인 병렬화 기법은 로드밸런싱 및 성능 확장성에 약점을 지니고 있다. DLP는 TLP처럼 각 프로세싱 모듈을 분리하지 않고 데이터를 나누어 병렬 처리한다. 따라서 프로세싱 모듈 간 수행 시간 차이로 인한 로드밸런싱 문제는 많이 줄어든다.

그러나 이를 위해서는 한 태스크가 한 번에 처리하는 데이터 단위(Data Unit)를 어떻게 나누느냐가 매우 중요한 요소이다. 표 2는 각 데이터 단위들을 비교한 것을 요약한 것이다.

GOP(Group Of Pictures) 단위로 병렬화하면 GOP 사이의 데이터 의존성이 전혀 없으므로 프로세서 코어가 늘어나는 만큼의 성능 향상을 기대할 수 있다. 그러나 GOP는 여러 프레임으로 구성되므로 메모리 요구사항이 크다. 프레임의 경우, I 프레임의 경우 데이터 의존성이 전혀 없으나, P 프레임이나 B 프레임의 경우 참조 프레임에 대한 데이터 의존성이 존재한다. 이러한 의존성으로 인해 동시에 수행할 수 있는 Thread의 개수가 제한 될 수 있다. 또한 프레임 단위로 처리되므로 메모리 요구사항도 비교적 크다. 슬라이스의 경우, 슬라이스 간 데이터 의존성이 없으나 한 프레임에 여러 개의 슬라이스가 존재하는 압축영상을 대상으로만 슬라이스 단위로 병렬 수행 할 수 있어서 사용성이 제한된다. 매크로블록단위의 경우 그림 13과 같이 매크로블록 사이의 데이터 의존성이 존재하여 성능 확장성이 좋지 않게 보이나 본 논문에서 제안하는 방법으로 성능 확장성을 높일 수 있고 메모리 요구사항도 작아 DLP를 위한 데이터 단위로 적합하다.

표 2. DLP를 위한 데이터 단위 비교
Table 2. Comparisons of Data Units for DLP.

Data Unit	Performance Scalability	Memory Requirements
GOP	High	Very Large
Frame	Low	Large
Slice	High	Depends on Size
MB	High	Low

2. HDLP: 수평적 DLP 기법

매크로블록단위의 DLP를 위해서는 의존성에 대한 세심한 검토가 필요하다. II장에서 설명한 바와 같이 기존의 연구들이 이 부분을 간과하여 H.264/AVC 표준에 정합하지 않는 결과를 만들어 낸다. 따라서 DF를 다른 프로세싱 단계와 묶어서 매크로블록 단위로 처리할 수 없다. 본 절에서 이들을 고려한 수평적 DLP 기법(HDLP: Horizontal DLP) 제안한다.

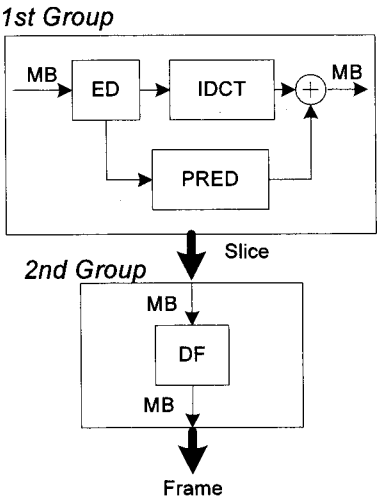


그림 14. 2 단계 프로세싱 그룹
Fig. 14. 2 Stage of Processing Groups.

HDLP는 그림 14에서와 같이 DF를 매크로블록 처리 그룹에서 분리하여 ED, IDCT, PRED의 그룹과 DF 그룹으로 나누어 처리한다. 즉, ED, IDCT, PRED로 하나의 슬라이스를 완성하면 이 슬라이스를 입력으로 하여 DF를 처리한다. 이때 DLP는 각각 프로세싱 그룹에 적용한다. 따라서 DF는 첫 번째 그룹에서 하나의 슬라이스가 완성될 때까지 시작할 수 없다. HDLP 기법의 쓰레드 스케줄링 방식은 다음의 pseudo 코드와 같다.

```
HDLP_schedule(thread th)
{
   /* Get macroblock line to be decoded */
   macroblock line mbl = get_next_macroblock_line();

  if no more macroblock line /* end of 1st group */
       goto DF processing /* go to 2nd group */
```

```
/* wait for completion of ED for previous
   macroblock line */
wait_for_complete_of_ED(mbl-1);

/* ED for a macroblock line */
ED(mbl);

for macroblock mb in mbl
{
    /* Get macroblock on which mb is dependent */
    macroblock mbd =
          get_dependent_macroblock(mbd);

    /* Wait for the completion of IDCT and
       PRED for mbd */
    wait_for_complete_of_idct_pred (mbd);

    /* IDCT and PRED for mb */
    IDCT_and_PRED(mb);
  }
}
```

그림 15는 위에서 설명한 쓰레드 스케줄링의 예를 보인 것이다. 쓰레드 Th1은 첫 번째 매크로블록 라인을 모두 디코딩한 후에 마지막 매크로블록 라인에 할당되어 ED을 진행하고 있으며, Th3는 ED를 끝내고 세 번째 매크로블록을 처리 중이다. Th4는 두 번째 매크로블록 처리를 시작하려고 하나, 그림 13의 의존성에 따라서 Th3가 세번 번째 매크로블록 디코딩을 끝나기를 기다리고 있다.

매크로블록 의존성은 그림 13에서 보는 바와 같이 주로 PRED와 DF 때문에 생긴다. HDLP에서는 DF를 두 번째 디코딩 그룹으로 분리하므로, PRED로 인한 의존성으로 기다림이 발생한다. 이것은 상위 라인의 디코딩 속도가 하위 라인 보다 느릴 때 발생하기 쉽다. HDLP

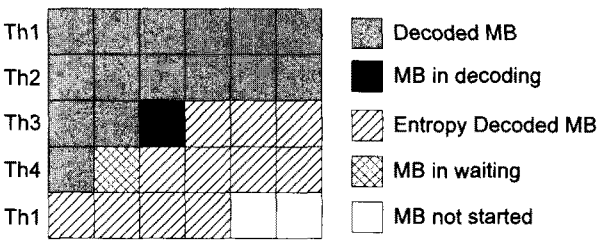


그림 15. HDLP 기법의 첫 번째 그룹 쓰레드 스케줄링
Fig. 15. Thread Scheduling of 1st Group in HDLP.

에서는 한 라인의 ED를 모두 끝낸 후에 다음 라인의 ED를 시작하게 함으로써 하위 라인의 시작 시간을 늦춤으로써 상위 라인 디코딩 시간과 하위 라인 디코딩 시간차를 늘리게 된다. 따라서 PRED 의존성으로 인한 기다림이 나타날 확률을 낮추게 된다.

매크로블록 단위로 쓰레드를 할당하게 되면, 매크로블록의 복잡도 차이로 인해서 로드밸런싱이 문제가 될 수 있다. 그러나 HDLP에서와 같이 매크로블록 라인 단위로 할당하게 되면 다양한 복잡도의 매크로블록들의 수행시간을 평균하는 효과가 있어 편차가 매크로블록 단위로 했을 때 보다 작아 로드밸런싱이 상대적으로 좋다.

또한 DF를 제외한 디코딩을 끝낸 후에 DF를 하게 함으로써, DF 수행 후의 값으로 prediction하지 않도록 하여 표준 정합성 문제가 없다.

3. HDLP 기법 성능 분석

수식적인 분석을 위해서 병렬화 이전의 H.264/AVC 비디오 디코딩 시간을 T_{org} 라 하고, 병렬화 기법 적용 후의 디코딩 시간을 T_{par} 라 하면, 이를 통해서 얻는 성능 향상치(Speed-up)인 S 는 수식 (1)로 나타낼 수 있다. 따라서 S 가 클수록 더 좋은 성능을 보인다고 할 수 있다.

$$S = \frac{T_{org}}{T_{par}} \tag{1}$$

i 번째 매크로블록 디코딩에서 ED에서 필요한 수행시간을 $t_e(i)$ 로, IDCT에서 필요한 시간을 $t_{idct}(i)$, PRED에서 필요한 시간을 $t_{pred}(i)$, 그리고, DF를 위해 필요한 시간을 $t_{df}(i)$라 한다면, 싱글 코어에서 단일 쓰레드로 동작시키는 환경에서, n개의 매크로블록으로 이루어진 H.264/AVC 비디오 디코딩에 필요한 시간 T_{org}는 수식 (2)와 같다.

$$T_{org} = \sum_{i}^{n} [t_e(i) + t_{idct}(i) + t_{pred}(i) + t_{df}(i)] \tag{2}$$

En: ED for n th row
Rn: IDCT & PRED for n th row
$t_{we}(n)$: Waiting Time before ED starts for n th row
$t_{wr}(n)$: Waiting Time in the middle of IDCT & PRED for n th row
t_{idle}: idle time

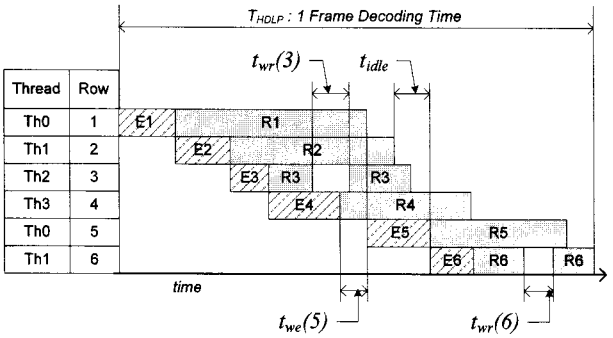


그림 16. 한 프레임에 대한 HDLP 스케줄링 및 디코딩 시간

Fig. 16. HDLP Scheduling and Decoding Time for a Frame.

본 절에서는 한 프레임에 디코딩 시간을 이용하여 성능 분석을 진행하는데, 1st Group에 대한 수행 시간으로 분석을 진행한다. 그림 16은 HDLP로 6개의 매크로블록라인을 갖는 한 프레임을 1st Group 작업을 수행할 때의 쓰레드 스케줄링 모습과 DF를 제외한 프레임 디코딩 시간을 보여준다. 따라서 본 절에서 사용될 T_{org}는 한 프레임의 디코딩 시간에서 DF 작업이 빠진 것으로 수식 (3)과 같다.

$$T_{org} = \sum_{i}^{n} [t_e(i) + t_{idct}(i) + t_{pred}(i)] \quad (3)$$

HDLP 스케줄링에서는 그림 16에 표시된 것처럼 다음과 같은 3가지 종류의 대기시간이 발생할 수 있다.

- $t_{wr}(n)$: n 번째 매크로블록 라인을 처리할 때, 그림 13에서와 같은 데이터 의존성으로 인해 이전 라인의 의존성을 갖는 매크로블록이 끝나기를 기다릴 때 발생하는 대기시간을 나타낸다. 그림 15의 4번째 라인의 격자무늬로 표시된 매크로블록이 그 예이다. PRED시 나타나는 데이터 의존성이므로, ED에서는 발생하지 않는다. 그림 16에서 $t_{wr}(3)$ 과 $t_{wr}(6)$는 매크로블록 라인 3과 6에서 위와 같은 이유로 발생하는 대기시간이며, En에서는 나타나지 않는다. 이 시간은 쓰레드 및 데이터 모두 대기하는 시간이다.
- $t_{we}(n)$: n 번째 매크로블록라인 처리를 시작할 때, 쓰레드가 이전에 할당된 매크로블록라인 처리가 끝나지 않아 발생하는 대기시간이다. 쓰레드 자체는 대기하는 것은 아니지만, 데이터가 처리되기를 기다리는 시간이다.
- t_{idle}: 할당된 쓰레드는 다른 일을 모두 마친 상태이나, 이전라인의 ED가 끝나지 않아 데이터가 처리를 시작하지 못하고 대기하는 시간이다. 그림 16에서 보면, Th1은 2번째 매크로블록라인 처리를 마치고 6번째 매크로블록라인 처리에 할당되었으나, 5번째 라인의 ED가 끝나지 않아 대기하는 시간이다.

표 1과 같이 Foreman의 t_r 시간이 Akiyo보다 상대적으로 크므로, Foreman은 그림 17의 (a)와 같이, Akiyo는 그림 17의 (b)와 같이 스케줄링될 것이다. 4개의 쓰레드 개수를 가지고 두 개 서로 다른 복잡도를 갖는 비트스트림을 HDLP로 디코딩하여 비교해 보면, 그림 17 (b)처럼 복잡도가 낮은 비트스트림의 t_{we}는 줄어드나, t_{idle}이 커진다.

따라서 쓰레드를 늘려서 t_{we}를 없애더라도 t_{idle}이 증가하게 되어 성능 향상이 좋지 않을 수 있다. 이를 좀

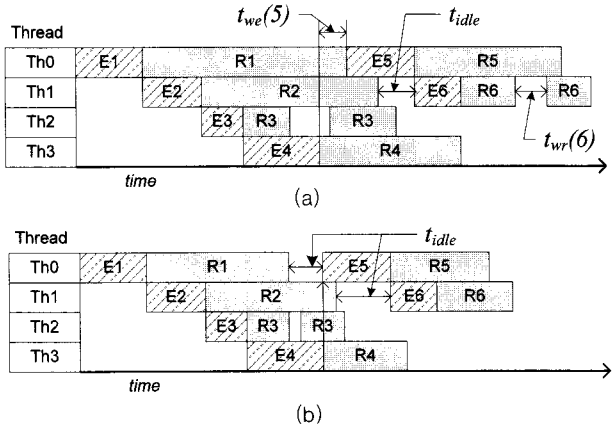


그림 17. 4개의 쓰레드를 사용하는 HDLP의 서로 다른 복잡도의 비트스트림 디코딩시 작업 스케줄링 비교

Fig. 17. Job Scheduling Comparison of HDLP with 4 Threads for Decoding of 2 Bitstreams with Different Complexity.

더 자세하게 분석하기 위해 발생 빈도가 낮은 t_{wr}이 없는 것으로 가정하고 Akiyo와 Foreman에 대해서 표 1에 있는 표준 수행시간을 이용하여 HDLP 디코더의 성능 확장성을 좀 더 자세히 분석해 본다. QCIF 크기의 비트스트림이므로 11개의 매크로블록을 갖는 9개의 매크로블록라인으로 구성된다.

t_e: ED execution time for an MB Line

t_r: IDCT & PRED execution time for an MB Line

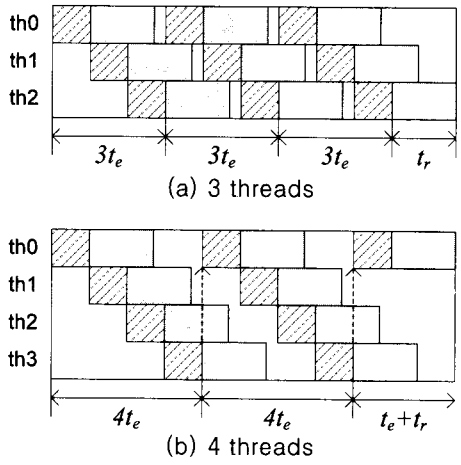


그림 18. Akiyo 표준화 실행시간을 이용한 높은 복잡도 비트스트림의 HDLP 스케줄링의 예

Fig. 18. An Example HDLP Scheduling of Lower-complexity Bitstream with Akiy's Normalized Execution Time.

그림 18은 표 1의 Akiyo 단계별 표준 실행 시간을 이용하여 모든 매크로블록 라인의 수행 시간이 표준 수행 시간을 갖는다는 가정 하에, 한 프레임을 HDLP로 디코딩할 때의 쓰레드 스케줄링 모습을 보인 것이다. (a)는 3개의 쓰레드를 사용할 때의 모습이고 (b)는 4개의 쓰레드를 사용할 때의 모습이다. (a)와 (b)의 한 프레임 디코딩 시간은 $9t_e + t_r$로 동일하다. (b)의 경우 복잡도에 비해서 쓰레드가 많을 경우의 모습을 보인 것으로, 작업 양에 비해서 쓰레드가 많아서 t_{idle}시간이 많은 상태이다. 따라서 쓰레드의 개수를 4개에서 3개로 줄이더라도 성능 차이가 없다. 그러나 더 줄이면 t_{we}가 증가하여 성능은 더 나빠질 것이다.

그림 19는 Foreman의 표준 수행 시간을 이용하여 그림 18과 같이 스케줄링 한 것으로, (a)는 3개의 쓰레드일 때, (b)는 4개의 쓰레드일 때, (c)는 5개의 쓰레드 일 때 HDLP 디코더로 한 프레임을 디코딩 할 때의 모습이다. (a)의 경우 쓰레드 수에 비해서 복잡도가 높은 비트스트림을 처리할 때의 모습을 보인 것과 같이 t_{we}가 크다. (b)는 쓰레드의 개수가 하나 늘어서 t_{we}가 (a)비해서 줄어들고 디코딩 시간도 t_{we}가 작아진 만큼 줄었다. (c)는 t_{we}가 0이 되고 t_{idle}시간이 생겼으나 디코딩 시간은 $9t_e + t_r + 2t_{we}$에서 $9t_e + t_r$로 줄어들었음을 보여준다.

그림 18과 19의 (c)는 t_{idle}이 발생하는 순간부터 쓰레드 개수를 늘리더라도 t_{idle}이 더 커져서 더 이상의

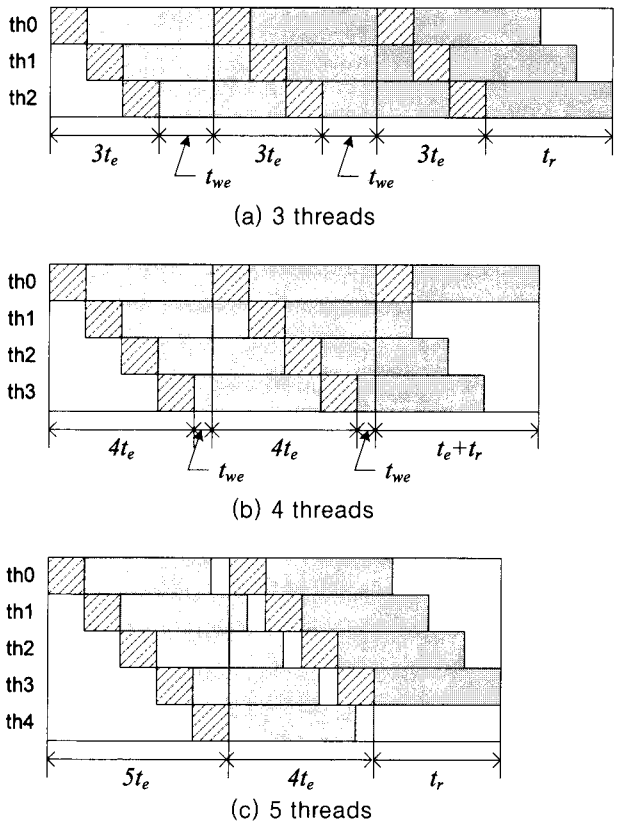


그림 19. Foreman 표준화 실행시간을 이용한 비트스트림의 HDLP 디코더 스케줄링의 예

Fig. 19. An Example of HDLP Scheduling of Higher-complexity Bitstream with Foreman's Normalized Execution Time.

성능 향상이 없음을 보여준다. 따라서 t_{we}가 0 일 때, 디코딩 시간이 가장 짧다. 이 때 디코딩 시간은 모든 라인의 ED시간의 합에 마지막 라인 R을 수행한 시간의 합으로 계산된다. 한 프레임 최소 디코딩 시간 T_{HDLP}는, N_l 을 한 프레임의 매크로블록 라인 개수로 했을 때 다음의 수식 (4)와 같이 표시 할 수 있다.

$$T_{HDLP} = N_l \times t_e + t_r \tag{4}$$

병렬화 이전의 디코딩 시간은 수식 (5)와 같으므로, 최대 성능 향상 치 $S_{\max}$는 수식 (6)과 같다.

$$T_{org} = N_l \times (t_e + t_r) \tag{5}$$

$$S_{\max} = \frac{T_{org}}{T_{HDLP}} = \frac{N_l \times (t_e + t_r)}{N_l \times t_e + t_r} \tag{6}$$

여기서 t_{we}가 0되어 최대의 성능을 보이기 위한 조건은 다음과 같다.

쓰레드가 처리해야 할 매크로블록 라인의 ED를 수행하기 위해서는 바로 이전 라인의 ED가 끝나야 한다.

그림 19의 (c)를 보면, th0가 첫 번째 라인을 처리한 후 다음 라인을 수행하기 위해서는 th4가 처리중인 ED가 끝나야 한다. 이것은 모든 쓰레드가 ED 처리를 마친 후를 의미하여, N_t 를 쓰레드 개수로 할 때, $N_t \times t_e$ 로 나타낼 수 있다. 그러나 이를 만족하더라도 쓰레드가 현재 라인의 R을 끝내지 못한다면 시작할 수 없다. 현재 라인 처리 시간은 $(t_e + t_r)$ 로 표현되므로 최대의 성능을 보이기 위한 조건은 수식 (7)로 표현할 수 있다.

$$N_t \times t_e \geq (t_e + t_r) \tag{7}$$

따라서 최대의 성능을 보일 때의 최소의 쓰레드 개수는 조건 식 (7)을 만족하는 최소의 N_t 이다.

t_{we}는 다음과 같은 식으로 나타낼 수 있다.

$$t_{we} = (t_e + t_r) - N_t \times t_e \tag{8}$$

따라서 t_{we}가 0보다 클 경우의 성능 향상치 S는 다음의 식 (9)와 같다.

$$S = \frac{N_l \times (t_e + t_r)}{N_l \times t_e + k t_{we} + t_r}, \quad where\ k = \left\lceil \frac{N_l}{N_t} \right\rceil - 1 \tag{9}$$

식 (8), (9)에서 쓰레드 개수 N_t가 클수록 t_{we}와 k가 작아져 성능향상이 더 많음을 예상 할 수 있다. 또한, 복잡하거나, HD 비디오처럼 프레임 사이즈가 클 경우 $(t_e + t_r)$이 커지므로, 조건 식 (7)을 만족하는 쓰레드 개수 커짐에 따라서 식 (8)에서와 같이 t_{we} 가 작아진다. 따라서 식 (9)로부터 쓰레드가 많을수록 성능 향상이 좋아짐을 알 수 있다. 따라서 위의 수식들로 복잡하고 데이터가 큰 비트스트림에서 성능 확장성이 좋을 것으로 예측 가능하다.

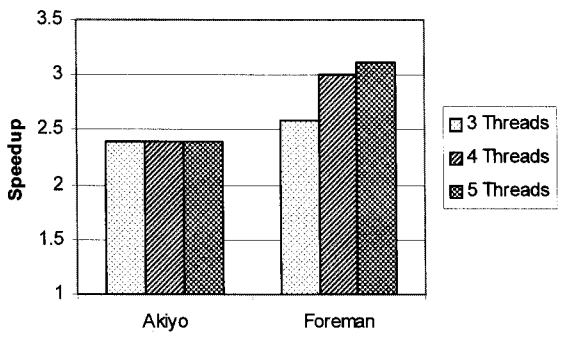


그림 20. HDLP 디코더의 Akiyo와 Foreman의 성능 예측
Fig. 20. HDLP Decoder's Expected Performance of Akiyo and Foreman.

그림 20의 그래프는 위의 성능 향상 식을 이용하여 예측한 Akiyo와 Foreman의 성능 확장성이다. Akiyo의 경우 쓰레드가 3개 이상에서는 더 이상의 성능 향상이 없고, 조금 더 복잡한 Foreman의 경우, 쓰레드의 개수가 4개까지 성능 향상이 크나, 5개부터 성능 향상이 작음을 보여 준다.

Ⅳ. 실 험

1. 실험환경

제안한 병렬화 기법에 대한 성능분석 내용을 검증하기 위해 JM 13.2을 사용했다. 병렬화 수행을 위해서 4개의 ARM11 코어가 204 MHz로 동작하는 ARM11 MPCore Evaluation 보드를 사용했다. 각 ARM11 코어는 32 KB의 L1 명령어/데이타 캐시를 가지며, 1 MB의 L2 캐시를가 있다. 이들 코어는 30MHz로 동작하는 AXI 버스로 연결된다[4].

본 논문에서는 병렬화를 위해서 OpenMP 모델을 사용했는데, 이를 위해 JM 13.2를 OpenMP 모델에 맞게 수정했다. 그러나 ARM11 MPCore를 위한 OpenMP 컴파일러가 없어, 멀티 쓰레드 C 코드를 생성하는 OMPi를 사용하여 C 코드로 변환한 후[13], GNU C 컴파일러 버전 4.2.0을 사용하여 최종적인 ARM11 MPCore 실행 코드를 생성했다.

정확한 분석을 위해, cycle-accurate 한 co-verification 검증 환경인 SoCDesigner[14]을 사용해서 정확한 프로파일링 결과를 추출했다. 앞에서 보인 프로파일링 결과는 모두 이러한 환경을 통해서 실험적으로 검출된 cycle-accurate한 결과이다.

2. 실험 결과 및 분석

그림 21은 4개의 쓰레드를 사용했을 때, HDLP 실험 결과와 TLP 실험 결과를 비교한 것으로, HDLP 실험 결과 예측한 것과 마찬가지로 로드밸런싱이 좋아 TLP 보다 더 좋은 성능을 보인다. 또한 HDLP가 복잡한 비

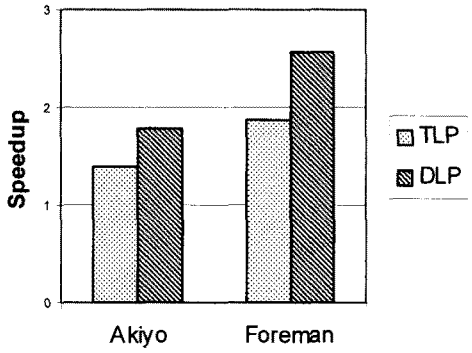


그림 21. HDLP 성능 측정 결과 및 TLP와의 성능비교
Fig. 21. HDLP Experiment Results for Performance Improvements and Comparisons with TLP.

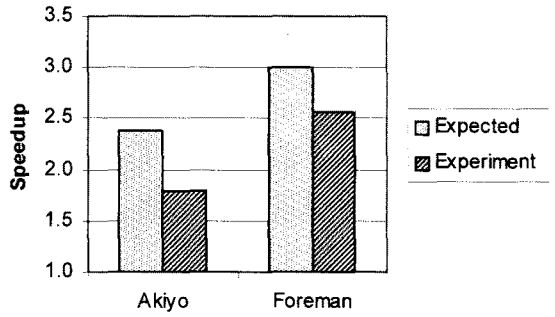


그림 22. HDLP 예측 성능과 측정 결과 비교
Fig. 22. Comparison of HDLP Expected Performance and Experiment Results.

트스트림에서는 Idle 시간이 작아 더 좋은 성능을 보일 것으로 예측한 대로 Foreman에서 더 좋은 성능을 보이고 있다.

그림 22는 예측된 성능과 실험 측정 결과를 비교한 것이다. HDLP 적용시 Akiyo의 경우 2.39 배, Foreman의 경우 3.0 배의 성능향상이 있을 것으로 예측되었다. 실험 결과 Akiyo, Foreman 각각 1.79와 2.56 측정되어 예측한 것과 차이를 보인다. 이것은 OpenMP 멀티쓰레드 동기화시 발행하는 오버헤드[16]와 메모리 입/출력 오버헤드 등을 고려하지 않아서 발생한 것으로 보인다.

메모리 오버헤드가 사용한 비디오 크기가 작아서 상대적으로 작다고 본다면, OpenMP 오버헤드만을 고려한 스케줄링은 그림 23에서 보이는 것과 같다. 즉, 쓰레드가 스케줄링되어 디코딩을 시작할 때 오버헤드 시간 t_{omp} 만큼 지연시간이 발생한다. 따라서 성능 향상 예측 식 (9)는 t_{omp}을 반영한 수식 (10)으로 바뀐다.

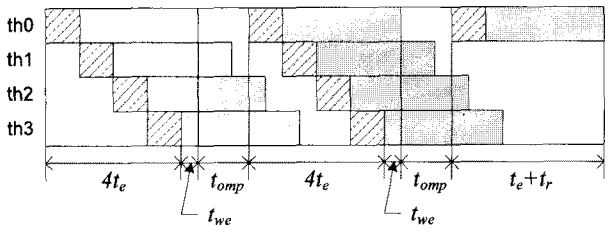


그림 23. OpenMP 오버헤드를 고려한 HDLP 쓰레드 스케줄링
Fig. 23. HDLP Thread Scheduling with OpenMP Overhead.

$$S=\frac{N_l\times(t_e+t_r)}{N_l\times t_e+k(t_{we}+t_{omp})+t_r} \qquad (10)$$
$$where\ k=\left\lceil\frac{N_l}{N_t}\right\rceil-1$$

OpenMP 오버헤드는 데이터 복잡도와는 관계없고 쓰레드 개수가 많아질수록 커지므로, 같은 쓰레드 개수에서 t_{omp}는 일정한 값을 갖는다고 볼 수 있다. 수식 (10)에서 복잡도가 낮은 비트스트림은 분자의 크기가 상대적으로 작고, t_{omp}는 상대적으로 크므로 성능 향상 S는 더 작아진다. 따라서 그림 23과 수식 (10)로 복잡도가 낮은 Akiyo의 성능 예측과 실험 측정 결과 차이가 더 커진 것이 OpenMP 오버헤드가 중요한 요인임이 설명된다.

그림 24는 QCIF사이즈의 BUS 비트스트림을 HDLP 디코더로 디코딩할 때 성능 확장성 측정한 실험 결과이다. 쓰레드 개수가 2 개에서 3개 일 때 성능 향상이 있으나, 4개일 때는 더 이상의 성능향상이 없다. 이는 Ⅲ장에서 분석한 것과 같이 Idle시간이 길어져 더 이상의 성능 향상이 어렵다는 것이 실험적으로 확인 된 것이다.

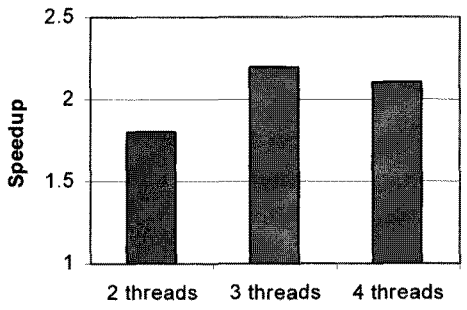


그림 24. HDLP 성능 확장성 측정 결과
Fig. 24. HDLP Experiment Results for Performance Scalability.

그림 20의 성능 예측 그래프를 보면 최대 쓰레드 개수가 늘어남에 따라서 성능 향상도 커지나 최대 성능을 내는 쓰레드 개수보다 많아지면 더 이상의 성능 향상이 없다. 그런데 그림 24에서는 쓰레드가 4개일 때 오히려 줄었다. 이는 OpenMP의 오버헤드가 쓰레드개수가 늘어남에 따라서 더 커지므로, 이로 인해서 오히려 성능이 감소한 것으로 분석할 수 있다.

V. 결 론

DLP는 로드밸런싱 및 성능 확장성에서 TLP보다 좋은 성능을 보이나, 데이터 의존성이 없도록 쓰레드에 할당할 데이터 분할이 잘되어야 한다. 본 논문에서는 기존 연구들이 간과한 H.264/AVC 표준 정합성을 만족하도록 DF를 분리한 두 개의 그룹으로 분리하여 PRED와 DF가 정확한 순서대로 실행되도록 한 수평적 DLP 기법인 HDLP 기법을 제안했다. HDLP는 매크로라인별로 쓰레드를 할당하며, 각 쓰레드는 매크로블록라인에 대한 ED 작업 후 나머지 작업을 수행한다. 제안된 HDLP의 성능 및 성능 확장성에 대해서 수식적으로 분석하여, 복잡한 비트스트림 및 큰 프레임 사이즈의 비트스트림에서 더 좋은 성능향상이 있을 것으로 예측했다. 또한 HDLP로 가장 좋은 성능을 내는 최소한의 쓰레드 개수를 어떻게 구하는 지에 대해서도 분석했다.

제안된 HDLP는 OpenMP 모델을 이용하여 H.264/AVC 참조 코덱 소프트웨어인 JM 13.2를 수정하여 구현하였으며, 이를 ARM11 MPCore환경에서 실행하여 얻은 실험 결과를 보면, 수식적으로 예측한 바와 같이 복잡한 비트스트림이 더 좋은 성능을 보였고, 쓰레드 개수가 일정한 수를 넘어 가면 더 이상의 성능 향상이 어려움도 실험적으로 나타났다. 다만, 수식적으로 분석된 성능 향상이 복잡도에 따라서 15% ~ 25% 정도 차이가 있었는데, 이것은 실제 실험환경에서 발생하는 OpenMP 오버헤드와 메모리 오버헤드 등을 수식에 반영하지 않은 결과로 분석된다. 특히 OpenMP 오버헤드는 복잡도에 관계없이 일정하므로 복잡도가 낮은 비트스트림에서 오버헤드가 상대적으로 커서 더 많이 차이를 보이고 있다.

분석에 사용된 ED, IDCT, PRED, DF 등의 표준화 수행 시간은 실제 환경에서 수행한 프로파일을 통해서 정확히 측정된 값을 이용하여 만들어졌다. 그러나 사용된 비트스트림의 개수가 작아서 일반화하기가 어려울 수도 있으나, 복잡도에 따른 표준 비트스트림들을 선정하여 성능 측정을 위한 표준 비트스트림별 표준화 값을 갖는 테이블을 통하여 좀 더 정확하고 다양한 형태의 예측 모델을 만들 수 있을 것으로 보인다. 또한 코어 사이 혹은 메모리사이에서의 데이터 트래픽을 고려하고, OpenMP 오버헤드에 대해서 좀 더 자세한 분석을 통하여 수식적 모델을 개선한다면 더 실제의 상황에 맞는 예측 모델을 만들 수 있을 것이다.

참 고 문 헌

[1] ITU-T Recommendation H.264, SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS Infrastructure of audiovisual services - Coding of moving video, May 2003.

[2] ISO, Information Technology-Coding of Audio-Visual Objects, Part10-Advanced Video Coding, ISO/IEC 14496-10.

[3] Thomas Wiegand, Gary J. Sullivan, Gisle Bjøntegaard, and Ajay Luthra, Senior Member, "Overview of the H.264/AVC Video Coding Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560-576, July 2003

[4] K. Hirata and J. Goodacre. "ARM MPCore: the streamlined and scalable ARM11 processor core," *ASP-DAC '07*, Jan. 2007.

[5] L. Dagum et al. "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Com. Sci.Eng.*, 5(1):46 - 55, 1998.

[6] Song Hyun Jo, Han Wook Cho, and Yong Ho Song, "The Software Optimization of an MPEG-2 Video Decoder for Embedded Systems," *Proceedings of International Conference on Ubiquitous Convergence Technology, ICUCT / IWUCT 2008*, pp. 90-95, Khabarovsk, Russia, August 2008.

[7] Michael Horowitz, Anthony Joch, Faouzi Kossentini, and Antti Hallapuro, "H.264/AVC Baseline Profile Decoder Complexity Analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 704-716 July 2003.

[8] Tom R. Jacobs, Vassilios A. Chouliaras Jose L. Nunez-Yanez, "A Thread and Data-Parallel MPEG-4 Video Encoder for a System-On-Chip Multiprocessor," *Proceedings of the16th International Conference on Application-Specific*

Systems, Architecture and Processors, Samos, Greece, July 2005.

[9] Yen-Kuang Chen, Xinmin Tian, Steven Ge, and Milind Girkar, "Implementation of H.264 Encoder and Decoder on Personal Computers," *Journal of Visual Communication and Image Representation*, vol. 17, no. 2, pp. 509-532, 2006.

[10] E.B. van der Tol, E.G. Jaspers, and R.H. Gelderblom, "Mapping of H.264 Decoding on a Multiprocessor Architecture," *Proc. SPIE Conf. on Image and Video Communications and Processing*, 2003.

[11] Cor Meenderinck, Arnaldo Azevedo, Mauricio Alvarez, Ben Juurlink1, and Alex Ramirez, "Scalability of H.264," *Proceedings of the 1stworkshop on Programmability Issues for Multi-Core Computers*, Goteborg, Sweden, January 2008.

[12] J. Chong, N. R. Satish, B. Catanzaro, K. Ravindran, and K. Keutzer, "Efficient Parallelization of H.264 Decoding with Macro Block Level Scheduling," *IEEE International Conference on Multimedia and Expo*, pp. 1874-1877, July 2007.

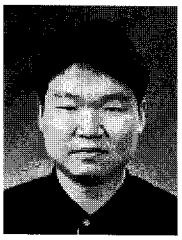
[13] JM Reference Software 13.2, http://iphome.hhi.de/suehring/tml/download/old_jm/jm13.2.zip

[14] V.V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable C compiler for OpenMP V.2.0," *in Proceedings of the EWOMP 2003, 5th European Workshop on OpenMP*, Sept. 2003.

[15] ARM-ESL SocDesigner, http://www.arm.com

[16] Chunhua Liao, Zhenying Liu, Lei Huang, and Barbara Chapman,"Evaluating OpenMP on Chip MultiThreading Platforms," *First International Workshop on OpenMP*, Eugene, USA, June, 2005.

저 자 소 개



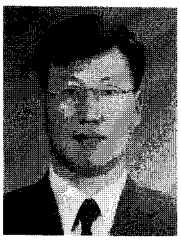
조 한 욱(학생회원)

1992년 서울대학교 컴퓨터공학과 학사 졸업.

1994년 서울대학교 컴퓨터공학과 석사 졸업.

2002년 University of Southern California, Electrical Engineering, 석사 졸업

<주관심분야 : 임베디드시스템, 멀티미디어, WLAN>



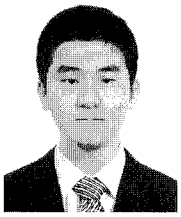
송 용 호(정회원)

1989년 서울대학교 컴퓨터공학과 학사 졸업.

1991년 서울대학교 컴퓨터공학과 석사 졸업.

2002년 University of Southern California, Electrical Engineering, 박사 졸업.

<주관심분야 : 임베디드시스템, SoC, NoC>



조 송 현(학생회원)

2006년 한양대학교 정보통신학부 학사 졸업.

2009년 한양대학교 전자컴퓨터통신공학과 석사 졸업.

<주관심분야 : 임베디드시스템, 멀티미디어>