공학박사학위논문

디스플레이 장치를 위한 고정 비율 압축 하드웨어 설계

Fixed Ratio Compression Hardware Design for Display Devices

2016 년 2 월

서울대학교 대학원

전기 컴퓨터 공학부

김 선 웅

초록

디스플레이 장치에서의 압축 방식은 일반적인 비디오 압축 표준과는 다른 몇 가지 특징이 있다. 첫째, 특수한 어플리케이션을 목표로 한다. 둘째, 압축 이득, 소비 전력, 실시간 처리 등을 위해 하드웨어 크기가 작고, 목표로 하는 압축률이 낮다. 셋째, 래스터 주사 순서에 적합해야 한다. 넷째, 프레임 메모리 크기를 제한시키거나 임의 접근을 하기 위하여 압축 단위당 목표 압축률을 실시간으로 정확히 맞출 수 있어야 한다. 본 논문에서는 이와 같은 특징을 만족시키는 세 가지 압축 알고리즘과 하드웨어 구조를 제안하도록 한다.

LCD 오버드라이브를 위한 압축 방식으로는 BTC(block truncation coding) 기반의 압축 방식을 제안하도록 한다. 본 논문은 압축 이득을 증가시키기 위하여 목표 압축률 12에 대한 압축 방식을 제안하는데, 압축 효율을 향상시키기 위하여 크게 두 가지 방법을 이용한다. 첫 번째는 이웃하는 블록과의 공간적 연관성을 이용하여 비트를 절약하는 방법이다. 그리고 두 번째는 단순한 영역은 2×16 코딩 블록, 복잡한 영역은 2×8 코딩 블록을 이용하는 방법이다. 2×8 코딩 블록을 이용하는 경우 목표 압축률을 맞추기 위하여 첫 번째 방법으로 절약된 비트를 이용한다.

저비용 근접-무손실 프레임 메모리 압축을 위한 방식으로는 1D SPIHT(set partitioning in hierarchical trees) 기반의 압축 방식을 제안하도록 한다. SPIHT은 고정 목표 압축률을 맞추는데 매우 효과적인 압축 방식이

다. 그러나 1D 형태인 1D SPIHT은 래스터 주사 순서에 적합함에도 관련 연구가 많이 진행되지 않았다. 본 논문은 1D SPIHT의 가장 큰 문제점인 속도 문제를 해결할 수 있는 하드웨어 구조를 제안한다. 이를 위해 1D SPIHT 알고리즘은 병렬성을 이용할 수 있는 형태로 수정된다. 인코더의 경우 병렬 처리를 방해하는 의존 관계가 해결되고, 파이프라인 스케쥴링이 가능하게 된다. 디코더의 경우 병렬로 동작하는 각 패스가 디코딩할 비트스트림의 길이를 미리 예측할 수 있도록 알고리즘이 수정된다.

고충실도(high-fidelity) RGBW 컬러 이미지 압축을 위한 방식으로는 예측 기반의 압축 방식을 제안하도록 한다. 제안 예측 방식은 두 단계의 차분 과정으로 구성된다. 첫 번째는 공간적 연관성을 이용하는 단계이고, 두 번째는 인터-컬러 연관성을 이용하는 단계이다. 코딩의 경우 압축 효율이 높은 VLC(variable length coding) 방식을 이용하도록 한다. 그러나 기존의 VLC 방식은 목표 압축률을 정확히 맞추는데 어려움이 있었으므로 본 논문에서는 Golomb-Rice 코딩을 기반으로 한 고정 길이 압축 방식을 제안하도록 한다. 제안 인코더는 프리-코더와 포스터-코더로 구성되어 있다. 프리-코더는 특정 상황에 대하여 실제 인코딩을 수행하고, 다른 모든 상황에 대한 예측 인코딩 정보를 계산하여 포스터-코더에 전달한다. 그리고 포스트-코더는 전달받은 정보를 이용하여 실제 비트스트림을 생성한다.

주요어 : liquid-crystal display(LCD), color image compression, real-time compression, VLSI implementation

학 번 : 2012-30197

목차

표 목차

그림 목차

제 1 장 서론

1.1 연구 배경

최근 디스플레이 장치의 해상도가 급격히 증가하고 있고, 이와 같은 추세가 한동안 지속될 전망이다. 일반적으로 디스플레이 장치는 다양한 목적을 위해 프레임을 저장하는 메모리를 가지고 있는데, 이와 같은 해상도 증가는 프레임 메모리 크기, 즉 하드웨어 가격에 직접적인 영향을 미치게 된다. 따라서 프레임 메모리 크기를 제한시키기 위한 프레임 메모리 압축 방식을 요구하는 디스플레이 장치가 점점 늘어나고 있다.

디스플레이 장치에서의 압축 방식은 일반적인 비디오 압축 표준과는 다른 몇 가지 특징이 있다. 우선 디스플레이 장치에서의 압축 방식은 특수한 어플리케이션을 목표로 한다. 가령, LCD 오버드라이브(overdrive)에서 이용되는 압축 방식의 경우 이미지의 화질보다는 가장자리(edge) 보존 특성을 더 중시한다. 그리고 디스플레이 장치에서의 압축 방식은 압축 이득, 소비 전력, 실시간 처리 등을 위해 하드웨어 크기가 작고, 목표로 하는 압축률이 낮다. 이 때문에 시간적 연관성(temporal correlation)은 활용되지 못하며, 이용되는 공간적 연관성(spatial correlation) 역시 제한적이다. 따라서 디스플레이 장치에서의 압축 방식은 상당히 제한된 상황에서의 이미지 압축 방식으로 분류할 수 있다. 또한 디스플레이 패널이 래스터 주사(raster scan) 순서로 입력 데이터를 전달받기 때문에 압축 방식 역시 이 순서에 적합해야 한다. 마지막으로 프레임 메모리 크기를 제한시키거나

임의 접근(random access)을 하기 위하여 압축 단위당 목표 압축률을 실시간으로 정확히 맞출 수 있어야 한다.

손실(lossy) 이미지 압축 알고리즘에는 다양한 방식들이 존재한다. 그림 1.1은 여러 손실 이미지 압축 알고리즘들을 압축 효율, 비반복적 레이트 조절(임의 접근), 하드웨어 비용(복잡도) 등의 기준으로 비교한 그래프를 보인다. 널리 알려진 JPEG[1]이나 JPEG2000[2]은 높은 압축 효율을 보이지만 하드웨어 비용이 매우 큰 문제가 있다. 앞서 언급한 바와 같이 디스플레이 장치에서의 압축 방식은 압축 이득, 소비 전력, 실시간 처리 등을 위하여 하드웨어 크기가 작아야 한다. 따라서 이와 같은 방식들은 디스플레이 장치에서의 압축 방식에 적합하지 않다.

한 블록 내 존재하는 픽셀들을 몇 개의 대푯값으로 표현하는 BTC(block truncation coding)는 장점과 단점이 분명한 알고리즘이다[3]. 우선 굉장히 단순한 방법으로 이미지를 압축하기 때문에 하드웨어 비용이 매우 낮다. 또한 압축 블록 크기가 결정되면 비트스트림 길이도 고정되기 때문에 고정 비트-길이를 정확하게 맞추는데 매우 유리하다. 또한 블록 내부의 튀는 값을 비교적 잘 보존하기 때문에 가장자리 보존 특성이 우수하다. 그러나 블록 내 존재하는 픽셀들을 단순히 몇 개의 대푯값으로 표현하기 때문에 에러가 상당히 크게 발생한다. 또한 블록 크기에 따라 고정 압축비가 결정되기 때문에 가능한 압축비가 제한되어 있고, 블록의 크기가 커질수록 화질이 급격히 떨어지는 단점이 있다.

SPIHT(set partitioning in hierarchical trees)은 고정 비트-길이를 맞추는데 적합한 알고리즘이다[4]. 또한 하드웨어 비용 대비 압축 효율이 매우 우수한 것으로 알려져 있다. 따라서 근접-무손실(near-lossless) 압축을 요구

하는 디스플레이 압축 방식에 적합하다. 그러나 보편적인 2D SPIHT은 정사각형 블록 단위의 압축을 수행하기 때문에 래스터 주사 순서에 적합하지 않고, 변형 형태인 1D SPIHT[5-7]은 압축 효율 등에서의 연구가 많이 부족한 상황이다. 또한 서로 다른 웨이블릿(wavelet) 밴드 레벨 사이에 존재하는 의존 관계로 인해 하드웨어 속도가 낮은 문제가 있다.

JPEG-LS[8] 등의 예측 기반 압축 방식들은 확장 가능성과 융통성이 매우 뛰어나다. 다시 말해, 특수한 어플리케이션을 지원할 수 있는 잠재력을 가지고 있고, 상황에 맞춰 압축 효율과 하드웨어 비용 사이의 적절한 지점을 선택할 수 있다. 그리고 디스플레이 패널의 래스터 주사 순서에도 적합한 방식이다. 그러나 예측 기반의 압축 방식들은 고정 비트-길이를 정확하게 맞추는데 어려움이 있다. 또한 픽셀간 존재하는 의존 관계로 인해 하드웨어 속도가 낮은 문제가 있다.

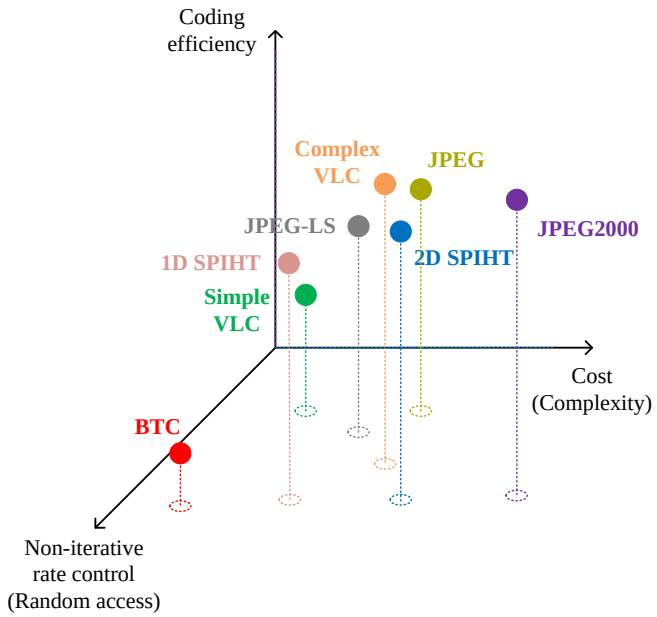


그림 1.1 손실 이미지 압축 알고리즘 비교

1.2 연구 내용

본 논문에서는 LCD(liquid crystal display) 장치의 세 가지 어플리케이션에 대한 압축 알고리즘과 하드웨어 구조를 제안하도록 한다. 첫 번째는 LCD 오버드라이브를 위한 압축 방식이다. LCD 오버드라이브는 액정(liquid crystal)의 늦은 반응 속도에 의한 모션 블러(motion blur)를 완화시키기 위한 방법으로 현재 프레임과 이전 프레임의 차이에 따라 서로 다른 전압을 부여하는 방식이다[9-11]. 이와 같은 동작을 수행하기 위해서는 이전 프레임을 메모리에 저장하고 있어야 하는데, 이 메모리 비용을 감소시키기 위해 그림 1.2와 같이 압축을 수행한다[12-14]. LCD 오버드라이브에서의 현재 프레임과 이전 프레임의 차이값은 특정 역치값(threshold)를 넘을 때에만 의미를 갖는다[15]. 또한 복원(reconstructed) 이미지는 참조 프레임으로 이용될 뿐 실제로 보여지는 이미지가 아니다. 따라서 전반적인 화질보다는 모션 블러가 크게 일어날 수 있는 가장자리 값을 보존하는 것이 더 중요하다. 그리고 압축 효율이 크게 높을 필요가 없기 때문에 압축률을 높이고 간단한 압축 알고리즘으로 구현하는 것이 압축 이득 측면에서 더 유리하다. 따라서 본 논문에서는 LCD 오버드라이브를 위한 압축 알고리즘으로 BTC[3]를 이용하도록 한다.

BTC는 기본적으로 2D 블록을 압축 단위로 이용한다. 그러나 이와 같은 압축 단위는 래스터 주사 순서를 이용하는 디스플레이 시스템에서 '이미지 너비×블록 높이'에 해당하는 라인 버퍼를 요구하게 된다. 또한 이 과정에서 지연 시간(latency) 역시 발생한다. 이와 같은 문제를 해결하기 위해서는 1D 블록을 이용하는 것이 가장 효과적이나 높이가 1인 블록에서는

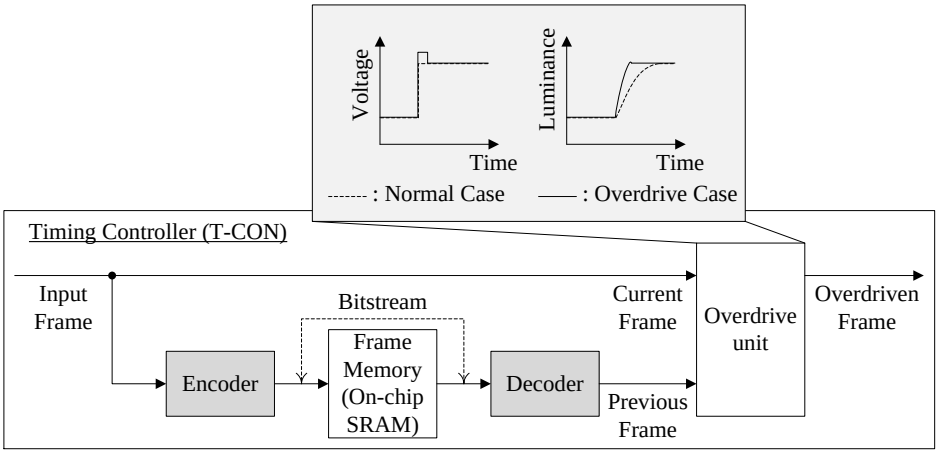


그림 1.2 LCD 오버드라이브를 위한 압축

감소된 공간적 연관성에 의해 화질 열화가 크게 발생하므로 라인 버퍼 크기와 화질 사이의 트레이드-오프 관계를 감안하여 압축 블록의 높이를 2로 설정하도록 한다. 또한 기존의 방식은 압축률 4 또는 6을 지원하는데, 최근 디스플레이 장치의 해상도 증가 추세를 감안했을 때 해당 압축률을 더 높일 필요가 있다. 따라서 본 논문에서는 화질 열화를 최소화시키면서 압축률 12를 지원하는 BTC 알고리즘을 제안하도록 한다. 제안 알고리즘은 실시간 처리를 위한 하드웨어 구조로 구현되었는데, 인코더와 디코더 모두 4-픽셀/사이클의 높은 처리 속도를 보인다. 또한 하드웨어 비용을 감소시킬 수 있는 프레임 메모리 인터페이스 등도 제안하도록 한다.

두 번째 어플리케이션은 저비용 근접-무손실 프레임 메모리 압축 방식이다. 디스플레이 장치에서는 프레임 레이트를 일정하게 유지시키지 못하는 경우가 가끔 발생한다. 이 경우 미처 준비되지 못한 현재 프레임 대신 이전 프레임을 출력하는데, 이 이전 프레임을 저장하기 위해 메모리가 이용된다. 이전 프레임을 압축할 때 고려해야 하는 사항은 다음과 같다. 우선

저장된 이전 프레임은 실제로 보여지는 프레임이므로 화질 열화가 거의 느껴지지 않아야 한다. 그러나 화질 열화를 막기 위해 복잡한 알고리즘을 이용할 경우 압축 이득이 감소하므로 비교적 간단한 알고리즘이어야 한다. 그리고 임의 접근을 위하여 블록 단위로 비트스트림 길이가 고정되어야 한다. 마지막으로 래스터 주사 순서를 지원하기 위해 1D 단위로 압축할 수 있어야 한다. 본 논문에서는 고정 비트-길이를 맞추는데 적합하고, 하드웨어 비용 대비 근접-무손실 압축에서 우수한 성능을 보이는 1D SPIHT[6]을 프레임 레이트를 유지시키기 위한 압축 알고리즘으로 이용하도록 한다.

기존의 1D SPIHT은 서로 다른 웨이블릿(wavelet) 밴드 레벨 사이에 존재하는 의존 관계로 인해 하드웨어 속도가 낮은 문제가 있었다[5-6]. 이 문제를 해결하기 위해 해당 의존 관계를 분석하고, 병렬 처리가 가능한 파이프라인 인코딩 구조를 제안하도록 한다. 디코더의 경우 비트스트림 내부에 존재하는 또 다른 의존 관계에 의해 처리 속도를 높일 수 없는 문제가 있었는데, 각 패스(pass)가 디코딩할 비트의 수를 미리 계산할 수 있도록 하드웨어 구조를 수정하여 인코더와 유사한 속도의 병렬 처리가 가능하게 한다.

세 번째 어플리케이션은 10 비트-너비 RGBW 컬러 도메인을 이용하는 디스플레이 장치를 위한 압축 방식이다. RGBW는 LCD 패널의 밝기를 향상시키기 위해 이용하는 컬러 도메인이다[16]. 백라이트 유닛의 밝기를 높이지 않고 패널의 밝기를 향상시킬 수 있기 때문에 소비 전력 등의 측면에서 각광받고 있다. 그러나 추가된 컬러 성분에 의해 전체 데이터 크기가 증가하기 때문에 프레임 데이터가 메모리에 저장될 경우 이전에 비해 더 큰 하드웨어 비용이 요구된다. 따라서 RGBW 컬러 도메인에서는

압축이 이전에 비해 훨씬 더 중요하다. RGBW 컬러 도메인을 이용하는 어플리케이션은 고해상도뿐만 아니라 고충실도(high fidelity)를 요구하는 경우가 많다. 이와 같은 특성은 전문적인 디지털 비디오 레코딩, 디지털 시네마, UHDTV, 메디컬 이미지 등 고가의 장치에서 주로 요구된다. 컬러 변환, 컬러 성분 서브-샘플링 등 컬러 이미지 압축에서 많이 이용되는 기법들은 원래의 컬러 성분을 변형시키고 컬러 성분간의 불균형 문제를 야기하므로 고충실도 이미지 압축에서는 일반적으로 제외된다[17-18]. RGBW 컬러 도메인의 경우 컬러 성분의 수가 4이기 때문에 이용할 수 있는 인터-컬러 연관성(inter-color correlation)이 매우 높다. 따라서 RGBW 컬러 도메인을 위한 압축 방식은 이 연관성을 잘 활용할 수 있어야 한다. 컬러 변환 기법이 제외된 주파수 도메인 기반의 압축 방식들은 이용할 수 있는 인터-컬러 연관성이 크지 않다. 따라서 본 논문에서는 확장성과 융통성이 뛰어난 예측 기반의 압축 방식을 RGBW 컬러 도메인 압축에 이용하도록 한다. 예측 기반의 압축 방식은 기본적으로 픽셀 단위의 압축 방식이기 때문에 비트-플레인(bit-plane) 단위의 압축 방식과는 달리 비트-너비가 증가해도 압축 속도에 크게 영향을 받지 않는 장점도 있다.

실제 디스플레이 장치의 경우 패널 종류에 따라 RGBW 컬러 도메인에서의 인터-컬러 연관성이 달라진다. 주로 비선형적 RGBW 데이터 값을 LUT에 저장한 후 사상(mapping)시키는 방식을 많이 이용한다[19-20]. 이는 RGB to RGBW 변환식에 비교적 무관한 일반적인 압축 방식이 필요하다는 사실을 의미한다. 따라서 본 논문에서는 최소한의 인터-컬러 연관성만을 이용하는 압축 알고리즘을 제안하도록 한다. 예측 기반의 압축 방식들은 주로 VLC(variable length coding) 방식을 이용하고, 이 방식은 고정

압축비를 맞추는데 어려움이 있다. 이 문제를 해결하기 위해 다양한 방법들이 제안되었으나 실시간 처리가 어렵거나 너무 많은 하드웨어 비용을 요구하는 단점이 있다. 본 논문에서는 두 번의 인코딩을 통해 비교적 간단하게 고정 압축비를 맞추는 방법을 제안하도록 한다. 그리고 높은 속도를 보이는 하드웨어 구조를 제안하도록 한다.

1.3 논문 구성

본 논문은 다음과 같이 구성되어 있다. 2장에서는 본 논문에서 이용하는 BTC, SPIHT, 예측 기반 코딩 등의 이전 연구 내용을 소개한다. 3장에서는 LCD 오버드라이브를 위한 BTC 알고리즘과 하드웨어 구조를 제안하도록 한다. 4장에서는 저비용 근접-무손실 프레임 메모리 압축을 위한 고속 1D SPIHT 알고리즘과 하드웨어 구조를 제안하도록 한다. 5장에서는 고충실도 RGBW 컬러 이미지를 압축하기 위한 고정 압축비 VLC 알고리즘과 하드웨어 구조를 제안하도록 한다. 3장, 4장, 5장은 각 알고리즘의 성능 분석과 하드웨어 구현 결과 분석 내용을 포함한다. 그리고 6장에서는 제안 방식들에 대한 추가 분석 내용을 보이도록 한다. 마지막 7장에서는 본 논문의 결론을 내리도록 한다.

제 2 장 이전 연구[1]

2.1 BTC

2.1.1 기본 BTC 알고리즘

BTC 알고리즘은 블록 크기에 따라서 압축률이 결정된다. 기본 BTC 알고리즘[3]은 4×4를 블록 크기로 이용하며, 각각의 블록은 서로 겹치지 않는다. 블록 내부의 픽셀들을 압축하기 위하여 두 개의 대푯값 (representative values; RV)이 계산되고, 블록 내부의 픽셀들은 두 개의 대푯값 중 하나의 값으로 사상된다. 따라서 각각의 픽셀은 하나의 비트로 표현이 된다. 블록 내 모든 픽셀을 표현하는 '한 비트'의 집합을 비트맵 (bitmap)이라고 한다. 비트맵을 생성하기 위해서는 블록 내 픽셀들의 평균값을 먼저 계산해야 한다. 식 (2.1)과 같이 블록 내 어떤 픽셀이 이 평균값보다 크다면 그 픽셀은 1로 표현된다. 반면, 블록 내 어떤 픽셀이 이 평균값보다 작거나 같다면 그 픽셀은 0으로 표현된다.

$$bitmap(n) = \begin{cases} 1, & if\ pixel(n) > mean \\ 0, & otherwise \end{cases} \quad (2.1)$$

식 (2.1)에서 *pixel(n)*은 해당 블록 내 n번째 픽셀을 의미한다. 블록 내 픽셀들은 계산된 비트맵에 따라 두 개의 그룹으로 나눠진다. 그런 다음 두 그룹의 평균값들이 각각 식 (2.2)와 식 (2.3)을 이용하여 계산된다. 계산된 평균값들은 각 그룹의 대푯값으로 이용되며, 본 논문에서는 각각 RV0와

[1] This chapter is part of articles presented in chapter 3 and chapter 4.

RV1으로 명명하도록 한다. 참고로 최초의 BTC 알고리즘에서는 RV0와 RV1 대신 평균값과 표준편차를 이용하였다[3]. 그러나 연산이 간단한 식 (2.2)와 식 (2.3)이 현재에는 많이 이용되고 있다[21-23].

$$RV0 = \frac{\sum pixel(n) \cdot \{1 - bitmap(n)\}}{\sum\{1 - bitmap(n)\}} \tag{2.2}$$

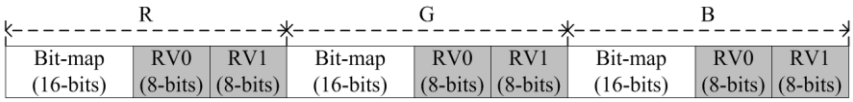
$$RV1 = \frac{\sum pixel(n) \cdot bitmap(n)}{\sum bitmap(n)} \tag{2.3}$$

한 블록 내부에서 계산된 비트맵과 대푯값들은 그 크기가 각각 16 비트와 2×8 비트이고, 최종 비트스트림에 포함되어 출력된다. 흑백 이미지의 경우 4×4 블록의 입력 데이터 크기는 4×4×8 비트이다. 따라서 기본 BTC 알고리즘의 압축률은 4(=4×4×8/(16+2×8))로 고정이 된다.

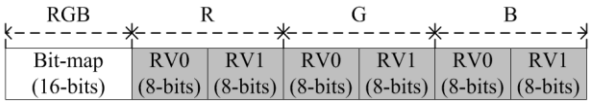
2.1.2 컬러 이미지 압축을 위한 BTC 알고리즘

기본 BTC 알고리즘은 컬러 이미지의 각 컬러 성분에 독립적으로 적용됨으로써 컬러 이미지를 압축할 수 있다. 그림 2.1 (a)는 이 방식의 비트스트림 포맷을 보인다. 비트스트림은 각 컬러 성분의 비트맵, RV0, RV1 등으로 구성된다. 이때의 블록당 비트스트림 길이는 96 비트이며, 압축률은 4이다. 압축률은 블록의 크기를 키워 증가시킬 수 있다. 그러나 두 개의 대푯값으로 표현되는 픽셀의 수가 많아지기 때문에 심각한 화질 열화 현상이 발생할 수 있다. 따라서 압축률을 높이기 위한 다른 형태의 접근 방법들이 연구되어 왔다.

단일 비트맵 BTC 방식은 네추럴 이미지에 존재하는 RGB 컬러 성분 간의 높은 인터-컬러 연관성을 이용한다[21]. 이 방식을 이용할 경우 단일 비트맵이 코딩되고, 세 컬러 성분이 이 단일 비트맵을 공유한다. 그림 2.1



(a)



(b)

그림 2.1 컬러 이미지 압축에서의 비트스트림 포맷.

(a) 기본 BTC (b) 단일 비트맵 BTC

(b)는 단일 비트맵 방식을 이용했을 때의 비트스트림 포맷을 보인다. 비트스트림은 16 비트 크기의 비트맵과 여섯 개의 8 비트 크기의 대푯값들로 구성된다. 따라서 압축률은 6(=3×4×4×8/(16+3×2×8))으로 증가된다.

단일 비트맵 BTC 방식은 세 개의 컬러 성분을 한꺼번에 표현하므로 클러스터링(clustering)의 성능이 복원 이미지의 화질에 매우 큰 영향을 미친다. 컬러 이미지 압축을 위한 벡터 양자화(vector quantizer; VQ)-BTC의 경우 블록 내부의 픽셀-벡터를 두 그룹으로 분리시키기 위해 K-means 클러스터링[24] 또는 PCA(principle component analysis)[25]를 이용한다[26-27]. 클러스터링이 완료되면 단일 비트맵이 생성되고, 두 개의 대푯값이 계산된다. 두 개의 대푯값은 식 (2.2)와 식 (2.3)을 통해 계산된다. 단, 식 (2.2)와 식 (2.3)에서 *pixel(n)*과 *RVm*은 각각 $[pixel(n)^R\ pixel(n)^G\ pixel(n)^B]^T$와 $[RVm^R\ RVm^G\ RVm^B]^T$ *(m=0, 1)*으로 수정된다. 기본 VQ-BTC[26]의 비트스트림 포맷은 그림 2.1 (b)와 동일하다. 따라서 기본 VQ-BTC의 압축률은 6이다. VQ-BTC에서의 결과 화질을 향상시키기 위하여 [27]은 대푯값과 실제값의 차이를 이용하여 대푯값의 에러를 개선하는 방식을 제안하였다.

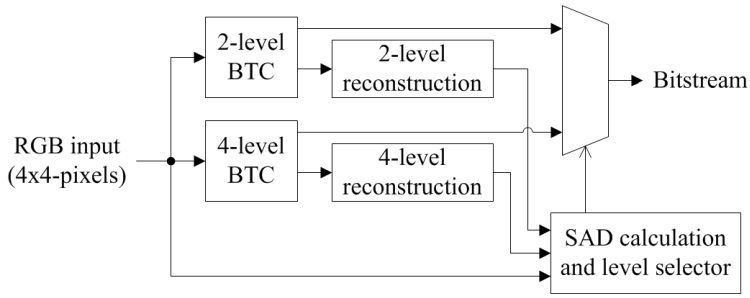


그림 2.2 AM-BTC의 구조

해당 방식은 대푯값의 에러를 감소시켜 기본 VQ-BTC의 성능을 향상시켰지만 클러스터링과 에러 개선 작업이 반복 연산을 비롯한 복잡한 연산을 요구하므로 실시간 처리가 어려운 문제가 있다.

선택적 다중-레벨(adaptive multi-level; AM)-BTC는 복잡한 영역을 정확하게 표현하기 위하여 2-레벨 BTC와 4-레벨 BTC 중 하나를 선택적으로 이용한다[28]. 이때, 2-레벨 BTC와 4-레벨 BTC의 압축률은 모두 6이다. 그림 2.2는 AM-BTC의 전체 구조를 보인다. 2-레벨 BTC와 4-레벨 BTC는 병렬로 수행되고, 실제 입력 데이터와 복원 데이터 사이의 SAD(sum of absolute differences) 값이 각 레벨마다 계산된다. SAD 값이 계산되면 더 작은 값을 갖는 레벨이 최종 코딩 레벨로 선택이 된다. AM-BTC는 2-레벨 BTC와 4-레벨 BTC를 복원 이미지의 에러를 기반으로 선택하기 때문에 화질 측면에 있어 매우 효과적이다. 하지만 AM-BTC는 픽셀 복원과 SAD 값을 위한 추가 계산과정을 요구한다. 또한 4-레벨 BTC는 목표 압축률을 맞추기 위해 4개의 대푯값 대신 하나의 평균값과 하나의 범위값을 코딩한다. 4-레벨 BTC 디코더는 식 (2.4)를 이용하여 픽셀값을 복원한다.

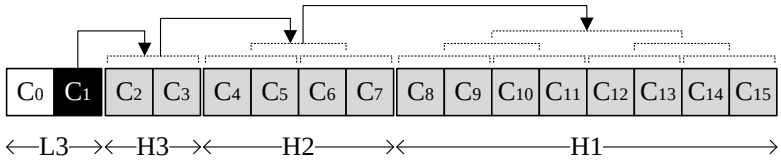
$$Recon.pixel = avg. + (2 \cdot bitmap(n) - 3)/6 \cdot range,$$
$$where\ bitmap(n) = 0, 1, 2, and\ 3 \tag{2.4}$$

4개의 대푯값을 코딩하는 일반적인 4-레벨 BTC와는 달리 기준이 되는 값이 하나이고, 이 값을 통해 4개의 대푯값이 식 (2.4)와 같이 계산되므로 AM-BTC에서는 복원 픽셀의 에러가 매우 클 수 있다. 특히 인터-컬러 연관성이 낮은 경우 잘못된 대푯값 계산으로 인해 에러가 매우 크게 발생할 수 있다. 게다가 압축률이 6 이상으로 설정될 경우 압축률을 맞추기 위해 기준이 되는 평균값에도 잘림(truncation) 연산을 적용해야 하므로 잘못된 대푯값 계산에서의 에러가 더 증가할 수 있다.

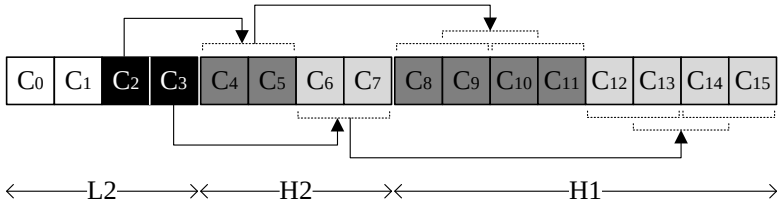
2.2 SPIHT

2.2.1 1D SPIHT 알고리즘

이 장에서는 1D SPIHT 알고리즘에 대해 설명하도록 한다. 우선 1D SPIHT에서 이용되는 자료 구조들에 대해 설명한 다음 구체적인 알고리즘 내용을 확인하도록 한다. 웨이블릿 도메인 기반의 압축 방식인 SPIHT은 DWT(discrete wavelet transform) 계수를 입력 데이터로 전달받는다. 전달받은 계수는 이진 트리 구조를 형성하고, 해당 구조는 중요성 검사(significance test)를 받는다. 계수 c_i는 두 개의 자식(children) 계수 c_{2i}와 c_{2i+1}를 갖는다. 그리고 계수 c_{2i}는 다시 두 개의 자식 계수 c_{4i}와 c_{4i+1}을 갖는다. 따라서 계수 c_{4i}, c_{4i+1}, c_{4i+2}, c_{4i+3} 또한 계수 c_i의 자손(descendant) 계수이다. 그림 2.3은 블록 크기가 1×16인 경우 1D SPIHT에서 이용되는 이진 트리의 모습을 보인다. 이진 트리는 DWT의 분해(decomposition) 레벨에 따라 다른 모양을 갖는다. 그림 2.3 (a)는 DWT 분해 레벨이 3일 때의 이진 트리를 보이고, 그림 2.3 (b)는 DWT 분해 레벨



(a)



(b)

그림 2.3 DWT 계수의 이진 트리. (a) 분해 레벨 3 (b) 분해 레벨 2

이 2일 때의 이진 트리를 보인다. 그림 2.3 (a)에서 계수 c_0와 c_1은 각각 레벨 3의 0번째와 1번째 저역 밴드 계수를 의미한다. 그리고 계수 c_2는 레벨 3의 0번째 고역 밴드 계수를 의미한다. 각 밴드 레벨에 존재하는 나머지 계수들은 그림 2.3 (a)에서 확인할 수 있다. 저역 밴드에 속하는 계수들 중 더 높은 인덱스를 갖는 절반의 계수들은 이진 트리의 루트(root)를 형성한다. 예를 들어 그림 2.3 (a)에서 c_1은 이진 트리의 유일한 루트이고, 그림 2.3 (b)에서 c_2와 c_3은 각각 두 이진 트리의 루트들을 의미한다.

기본 SPIHT은 list of insignificant sets(LIS), list of insignificant pixels(LIP), list of significant pixels(LSP) 등 세 가지의 자료 구조를 이용한다. 각각의 이진 트리는 내부 계수들이 모두 insignificant인 경우 LIS에 포함된다. 이 경우 이진 트리의 루트가 해당 집합을 대표하여 LIS에 포함된다. 중요성 검사가 진행될수록 이진 트리는 더 작은 이진 트리로 나누어지고, 나누어진 이진 트리에 대해 다시 검사를 수행한다. 중요성 검사가 계수(또는 픽

셀) 단위까지 내려올 경우 해당 픽셀은 LIP 또는 LSP에 포함된다. 만약 해당 계수가 중요성 검사를 통해 insignificant라고 판단되면 LIP에 속하게 되고, 그렇지 않으면 LSP에 속하게 된다. 코딩이 시작될 때 모든 저역 밴드 계수들은 LIP에 속하게 되고, 모든 이진 트리의 집합은 LIS에 속하게 된다. 예를 들어 그림 2.3 (a)에서 c_0와 c_1은 LIP에 속한 상태로 코딩을 시작하게 된다. 반면에 이진 트리의 루트인 c_1은 LIS에 속한 상태로 코딩을 시작하게 된다. 그림 2.3 (b)의 경우 c_0, c_1, c_2, c_3이 LIP에 속하며, 두 이진 트리의 루트인 c_2와 c_3가 LIS에 속한다.

SPIHT은 비트-플레인 기반의 이미지 압축 방식이다. 따라서 비트-플레인 단위로 압축이 수행되며, 높은 비트-플레인에서 낮은 비트-플레인 순서로 진행된다. 그림 2.4는 n번째 비트-플레인을 코딩하는 SPIHT 알고리즘을 보인다. SPIHT은 웨이블릿 계수들에 대하여 중요성 검사를 수행한다. 그림 2.4에서 집합 $O(i)$는 자식 계수(바로 아래 자손 계수)들의 집합을 의미하고, 집합 $D(i)$는 자손 계수들의 집합을 의미한다. 그리고 집합 $L(i)$는 $D(i) - O(i)$로 정의한다. $S_n(\cdot)$은 n번째 비트-플레인에 대한 중요성 검사를 의미하는데, 집합 내부의 가장 큰 계수가 해당 비트-플레인의 significance보다 같거나 크면 그 집합을 significant 집합으로 분류한다. 반면에 insignificant 집합으로 분류될 경우 해당 집합은 부호 '0'으로 코딩이 된다. 이 부분에서 상당수의 압축이 이뤄진다. Significant로 판단된 집합은 더 작은 하위 집합으로 나눠지고, 각각의 하위 집합은 다시 중요성 검사를 받게 된다. 만약 하위 집합이 하나의 계수만을 갖게 되면 그 집합은 중요성 검사 결과에 따라 significant 픽셀 또는 insignificant 픽셀로 구분이 된다. 그림 2.4의 SPIHT 알고리즘은 refinement pass(RP), insignificant pixel

Refinement Pass (RP)

1: **for** $each\ entry\ (i)\ in\ LSP$
2: **output** $magnitude\ of\ c_i$ # magnitude bit

Insignificant Pixel Pass (IPP)

3: **for** $each\ entry\ (i)\ in\ LIP$
4: **output** $S_n(i)$ # magnitude bit
5: **if** $S_n(i) = 1$ **then**
6: $\boldsymbol{move}\ (i)\ to\ LSP$
7: **output** $sign\ of\ c_i$ # sign bit

Insignificant Set Pass (ISP)

8: **for** $each\ entry\ (i)\ in\ LIS$
9: **if** $entry(\mathrm{i})\ is\ of\ Type\ A$ **then**
10: **output** $S_n(D(i))$ # sorting bit
11: **if** $S_n(D(i)) = 1$ **then**
12: **for** $each\ entry\ (j) \in O(i)$
13: **output** $S_n(j)$ # magnitude bit
14: **if** $S_n(j) = 1$ **then**
15: $\boldsymbol{add}\ (j)\ to\ LSP$
16: **output** $sign\ of\ c_j$ # sign bit
17: **else then**
18: $\boldsymbol{add}\ (j)\ to\ LIP$
19: **if** $L(i) \neq \emptyset$ **then**
20: $\boldsymbol{move}\ (i)\ to\ Type\ B$
21: **else then**
22: $\boldsymbol{remove}\ (i)\ from\ LIS$
23: **if** $entry(i)\ is\ of\ TypeB$ **then**
24: **output** $S_n(L(i))$ # sorting bit
25: **if** $S_n(L(i)) = 1$ **then**
26: **for** $each\ entry\ (j) \in O(i)$
27: $\boldsymbol{add}\ (j)\ to\ the\ LIS(Type\ A)$
28: $\boldsymbol{remove}\ (i)\ from\ the\ LIS$

그림 2.4 SPIHT 인코딩 알고리즘

pass(IPP), insignificant set pass(ISP) 등 총 3개의 패스로 구성된다. 참고로 최초의 SPIHT 알고리즘은 IPP와 ISP를 하나의 패스(sorting pass; SP)로 취급하였다[4].

2.2.2 SPIHT 하드웨어

기본 SPIHT 알고리즘은 웨이블릿 계수를 동적 순서(dynamic order)로 코딩한다. 이 경우 코딩 내용에 따라 코딩 순서가 변하기 때문에 메모리 이용 방법이 매우 복잡하게 되고, 결국 수행 속도가 느려지게 된다. 이 문제를 해결하기 위해 no-list SPIHT(NLS)이 제안되었다[29]. NLS는 동적 데이터 구조 대신 너비-우선(breadth-first) 순서를 이용하는 방식으로 수행 속도를 향상시켰다. [30]은 NLS의 하드웨어 구조를 제안하였는데, 최대 자손 계수값(maximum descendant magnitude value)을 미리 계산하고 활용하는 방식을 이용하여 메모리 요구량을 감소시켰다. 하지만 [30]은 병렬성을 효율적으로 이용하지 못하여 상대적으로 낮은 처리 속도(0.092 bit per cycle)를 보인다. [31]은 소비 전력 기반 SPIHT 코덱을 제안하였다. 해당 방법은 4개의 트리 파이프라이닝 구조를 이용하며, 4×4 블록의 한 비트-플레인을 한 사이클 동안 처리한다. 이때, 고정 코딩 순서가 적용되지만 4×4 블록 내부의 계수들은 직렬적으로 처리된다. 즉, 픽셀 병렬성이 전혀 이용되지 못한다. 따라서 해당 방법은 조합 논리 회로(combinational logic circuit)에서 긴 임계 경로(critical path) 시간을 갖고, 그 결과 동작 주파수는 10 MHz로 제한된다. 이때의 처리 속도는 CIF 4:2:0 이미지를 30 fps로 처리하는 아주 낮은 수준이다. [32]는 또 다른 고정 순서 SPIHT 인코더를 제안하였다. 해당 구조는 2×2 블록을 병렬적으로 인코딩하며, 각각의 비트-플레인은 파이프라인 방식으로 처리된다. 따라서 $N \times N$ 블록이 $N^2/4$ 사이클 동안에 인코딩된다. 이는 매우 높은 처리 속도이다. 그러나 [32]는 처리 속도를 높이기 위한 디코더 구조를 제안하지 않았다. 디코더의 경우 각각의 비트-플레인에 해당하는 비트스트림을 미리 나눌 수 없다. 따라서 인코더에서

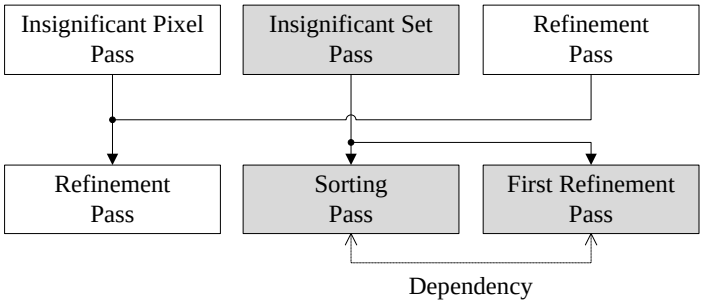


그림 2.5 BPS의 패스 재조직

이용한 방법을 그대로 적용하는 것이 불가능하다. [33]은 인코더와 디코더 모두에서 빠른 수행 속도를 보이는 BPS(block-based pass-parallel SPIHT) 구조를 제안하였다. 해당 구조는 4×4 블록의 한 비트-플레인을 한 사이클에 처리하는 속도를 보인다. BPS는 효율적인 병렬 처리를 위해 그림 2.5와 같이 패스를 재조직한다. 그 결과, 비교적 연산이 비슷한 RP와 IPP가 하나의 패스로 묶이고, 서로 다른 연산을 포함하고 있던 ISP가 SP와 FRP(first refinement pass) 등 두 개의 패스로 나눠진다. BPS 디코더는 각 패스에서 이번 사이클에 디코딩할 비트스트림의 길이를 미리 계산하고, 세 패스가 병렬적으로 동작할 수 있게 한다. 따라서 디코딩 속도가 인코딩 속도와 거의 동일하며, 이전 디코딩 하드웨어 구조들에 비해 더 높은 속도를 보이게 된다. 하지만 밴드 레벨 간의 의존성 문제 때문에 BPS의 블록 크기를 키울 수 없고, 그 결과 추가적인 처리 속도 향상은 불가능하다.

2.3 예측 기반 코딩

2.3.1 예측 방법

일반적으로 네추럴(natural) 이미지에서는 이웃하는 픽셀들이 강한 공간적 연관성을 갖는다. 따라서 주변 픽셀값들을 이용하여 참조값을 만든 다음 실제 입력값과의 차분값(또는 예측 에러)을 계산하여 코딩하는 예측 기반 코딩(predictive coding) 방법들이 많이 이용되고 있다. 예측 기반 코딩 방법은 크게 두 가지 부분으로 구성된다. 첫 번째는 주변 픽셀값들을 이용하여 참조값을 생성하는 예측기(predictor)이고, 두 번째는 예측 에러를 실제로 코딩하는 코더이다.

예측기는 간단한 방법부터 복잡한 방법까지 알고리즘에 따라 다양하게 존재한다. 간단한 DPCM의 경우 이웃하는 좌측 픽셀이나 상측 픽셀을 참조 픽셀로 이용한다[34]. JPEG-LS의 경우 식 (2.5)를 이용하여 참조 픽셀 x를 계산한다[8]. 참고로 식 (2.5)는 MED(median edge detection) 예측기 또는 LOCO-I 예측기라 불린다. VESA DSC(display stream compression) 표준도 수정된 MED 예측기를 이용한다[35].

$$x = \begin{cases} \min(A,B), & if\ C \geq \max(A,B) \\ \max(A,B), & if\ C \leq \min(A,B) \\ \mathrm{A} + \mathrm{B} - \mathrm{C}, & otherwise \end{cases} \tag{2.5}$$

식 (2.5)에서 A, B, C 픽셀은 각각 좌측, 상측, 좌상측 픽셀을 의미한다. 이와 같은 방법들 외에도 많은 예측 모드를 두고 가장 적절한 모드를 선택하는 예측 방법들이 많이 이용되고 있다[36-37].

예측 기반 코딩의 예측 방법은 확장 가능성이 매우 높다. 다시 말해, 특정 상황에 적합한 예측기를 만들 수 있는 장점이 있다. 또한 압축 효율

과 계산량 사이의 적절한 지점을 선택할 수 있는 장점도 있다. 따라서 많은 네추럴 이미지 압축 방식들이 예측 기반 코딩 방법을 이용하고 있다.

2.3.2 VLC

예측 기반 코딩 방법들은 VLC를 코더로 이용하는 경우가 많다[8, 35-36, 38-39]. 가령, JPEG-LS는 Golomb 코더를 이용하여 예측 에러를 코딩한다[8]. 그리고 VESA DSC 표준은 엔트로피 코더를 이용한다[35]. 본 연구에서는 하드웨어 구현이 비교적 간단하면서 높은 압축 효율을 보이는 Golomb-Rice 코더를 이용한다. 따라서 이 장에서는 VLC를 대표하여 Golomb-Rice 코더를 설명하도록 한다.

Golomb-Rice의 입력값은 예측 에러값이다. Golomb-Rice 코더를 수행하기 위해서는 입력값이 양수여야 한다. 따라서 식 (2.6)과 같이 예측 에러값을 변환하도록 한다.

$$source(n) = \begin{cases} 2|error(n)| & , \quad error(n) \geq 0 \\ 2|error(n)| - 1, & \quad error(n) < 0 \end{cases} \tag{2.6}$$

변환된 값 $source(n)$ 은 피제수로 이용되며, 제수 k에 의해 나눠진다. Golomb-Rice 코더는 나눗셈 결과 생성되는 몫과 나머지를 최종 비트스트림에 포함한다. 이때, 몫은 단항(unary) 코드로 표현되고, 나머지는 k 비트 크기의 이진(binary) 코드로 표현된다. 예를 들어 *error(n)*이 -29, k가 4인 경우 식 (2.6)을 통해 *source(n)*은 57이 된다. 그런 다음 57은 16(=2^4)으로 나뉘어져 몫과 나머지가 각각 3과 9가 된다. 3에 대한 단항 코드가 0001이고, 9에 대한 4 비트 크기 이진 코드가 1001이므로 생성되는 비트스트림은 00011001이다.

Golomb-Rice 코딩 자체는 무손실 압축 방식이며, 가변 길이 압축 방식이다. 따라서 고정 압축비가 주어진 상황에서는 비트스트림 길이를 제어하기 위한 방법이 추가적으로 필요하다. VLC 기반의 압축 방법들은 양자화 과정을 통해 비트스트림 길이를 제어한다. 가령, QL(quantization level)을 1만큼 올리게 되면 1-LSB(least significant bit)가 제거되면서 코딩해야 할 데이터의 크기 자체가 작아진다. 따라서 비트스트림 길이도 짧아지게 된다. QL을 조절하는 방법은 압축 알고리즘마다 다르다. [38]은 반복 연산을 통해 고정 압축비를 맞출 수 있는 QL을 결정한다. 즉, 그림 2.6과 같이 압축을 수행한 후 생성된 비트스트림의 길이가 목표 길이보다 긴 경우 QL을 1만큼 올려 다시 압축을 수행한다. 그러나 이와 같은 방법은 이미지의 복잡도에 따라 연산 시간이 변하게 되므로 실시간 처리가 어려운 문제가 있다. 모든 QL을 병렬로 처리한 후 조건을 만족하는 최소의 QL을 선택할 수도 있지만 이 방식은 8 비트-너비 데이터의 경우 8개의 인코더를 필요로 하는 문제가 있다. [35]와 [36]은 고정 압축비를 맞추기 위하여 비트-레이트 조절 회로를 따로 가지고 있다. 즉, 현재까지 코딩된 비트 수를 기반으로 QL을 조절하는 방식이다. 이때, 버퍼의 상황이나 레이트-왜곡(rate-distortion; RD) 수준을 고려하여 최적의 QL을 결정하기도 한다[35]. 그러나 이와 같은 방법들은 고정 비트스트림 길이를 정확하게 맞추는데 여전히 어려움이 있으며, 비트-레이트 조절 회로의 하드웨어 비용 역시 매우 큰 편이다. 따라서 고정 비트스트림 길이를 더 정확하게 맞추면서 하드웨어 비용도 낮은 구조가 필요하다.

Golomb-Rice 코딩의 장점 중 하나는 피제수와 제수값을 알고 있으면 생성되는 비트스트림의 길이를 식 (2.7)과 같이 쉽게 계산할 수 있다는

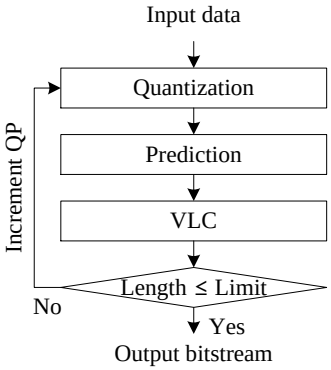


그림 2.6 고정 압축비를 위한 반복 양자화 방법

점이다. 본 논문은 이 특성을 이용하여 VLC의 약점인 고정 비트-길이 문제를 해결하고자 한다.

$$length_{GR} = \lfloor source/2^k \rfloor + k + 1 \qquad (2.7)$$

2.3.3 예측 기반 코딩 하드웨어

예측 기반 코딩 하드웨어의 경우 간단한 구조에서부터 복잡한 구조까지 다양하게 연구되어 왔다. [40]은 JPEG-LS에 대한 효율적인 파이프라인 구조를 제안하였다. 해당 구조는 기존 JPEG-LS 하드웨어 방식들에 비해 5배 이상의 빠른 속도를 보인다. 그러나 다른 예측 기반 코딩 하드웨어에 비해서는 여전히 처리 속도가 낮으며, 디코더를 위한 구조가 제안되지 않은 문제가 있다. 또한 하드웨어를 가볍고 빠르게 만들기 위해 근접-무손실 압축 부분을 제외한 문제가 있다. [34]는 하드웨어 구현에 유리한 예측 방식과 코딩 방식을 제안하였다. HACP(Hierarchical average and copy prediction)라 불리는 예측 방식의 경우 블록 내부에서 여러 개의 평균 예측값을 이용하여 더 정확한 예측을 가능하게 하였다. 그리고

SBT(significant bit truncation)라 불리는 코딩 방식은 해당 블록에서 코딩되는 비트 수를 미리 계산할 수 있게 하여 높은 처리 속도를 보일 수 있게 하였다. 그러나 비트 수를 미리 계산할 수 있도록 더미(dummy) 비트를 많이 포함시키므로 다른 예측 기반 코딩 방식에 비해 압축 효율이 낮다. 또한 손실 압축에 대한 부분이 존재하지 않기 때문에 해당 구조를 이용할 수 있는 범위가 좁다. [36]은 AGPM(associated geometric-based probability model)을 이용하여 병렬 처리를 방해하는 컨택스트(context) 테이블 없이 컨택스트 모델링이 가능하게 하였다. 그리고 상황에 따라 Golomb-Rice 코딩 또는 이진 코딩을 선택적으로 이용하는 방법을 제안하였다. 해당 방법은 JPEG-LS에 비해 더 적은 연산량으로 비슷한 결과를 보인다. 그러나 손실 압축을 위해 제안된 레이트 조절 방식이 고정 비트스트림 길이를 정확하게 맞추지 못하며, 디코딩이 특수한 처리 순서를 요구하는 문제가 있다. 또한 한 픽셀당 출력되는 헤더 비트가 압축률이 높아질수록 압축 효율에 악영향을 미치는 문제도 존재한다. [37]은 다양한 이미지 특성에 대응할 수 있는 MDA(multi-mode DPCM and averaging) 예측 방식과 차분값(residual)을 그룹핑하여 압축 효율을 향상시키는 방식을 제안하였다. 그리고 SFL(semi-fixed length) 코딩 방식을 통해 비트스트림의 길이를 미리 예측할 수 있게 하였다. 그 결과 매우 높은 하드웨어 속도를 보이게 되었다. 그러나 [34]와 마찬가지로 손실 압축을 지원하지 못하며, 비트스트림의 최대 17.8%가 헤더 비트로 이용되므로 목표 압축률이 높아질수록 압축 효율이 떨어지는 문제가 있다.

제 3 장 LCD 오버드라이브를 위한 BTC[2]

3.1 제안 알고리즘

이 장은 기존 단일 비트맵 BTC에 비해 2배의 압축률을 보이는 새로운 BTC 알고리즘을 제안한다. 래스터 주사 순서를 갖는 디스플레이 시스템에서의 라인 버퍼 크기를 감소시키기 위해서는 코딩 블록의 높이가 작아야 한다. 코딩 블록의 높이가 1인 경우 낮은 공간적 연관성에 의해 화질 열화가 크게 발생하므로 본 논문에서는 코딩 블록의 높이를 2로 설정하도록 한다. 기존 단일 비트맵 BTC의 2배 압축률인 12를 달성하기 위해서는 블록의 너비가 16(압축률 9.6) 또는 32(압축률 13.7)이어야 한다. 본 논문에서는 블록의 너비를 16으로 정하고, 압축률 12를 맞추기 위한 새로운 방식을 제안한다.

제안 방식은 기본 2×16 블록을 코딩하기 위하여 하나의 2×16 코딩 블록 또는 두 개의 2×8 코딩 블록을 선택적으로 이용한다. 2×16 코딩 블록은 가로로 긴 형태이며, 상대적으로 그 크기가 크다. 따라서 압축률이 높고 화질은 낮다. 반면에 두 개의 2×8 코딩 블록을 이용한 압축은 높은 화질을 보이지만 압축률은 낮다. 기존의 4-레벨 BTC는 픽셀당 2 비트 크기의 비트맵을 생성한다. 그러나 두 개의 2-레벨 2×8 코딩 블록은 픽셀당 1 비트 크기의 비트맵을 생성한다. 따라서 높은 압축률을 맞추는데 더 유

[2] This chapter is part of an article, "A Block Truncation Coding Algorithm and Hardware Implementation Targeting 1/12 Compression for LCD Overdrive," will be published in the IEEE Journal of Display Technology (accepted, October 10, 2015, DOI: 10-1109/JDT.2015.2493163).

리하다. 뿐만 아니라, 가로로 긴 블록의 경우 정사각형 블록에 비해 공간적 연관성이 더 낮다. 따라서 하나의 4-레벨 2×16 코딩 블록보다는 두 개의 2-레벨 2×8 코딩 블록이 압축에 더 효율적이다.

이어지는 장에서는 공간적 연관성을 이용하여 코딩되는 비트스트림의 길이를 감소시키는 방식을 제안한다. 절약된 비트 수는 두 개의 2×8 코딩 블록을 이용할 수 있게 하고, 그 결과 전반적인 화질이 향상된다.

3.1.1 비트-절약 방법

기본 2×16 블록이 하나의 2×16 코딩 블록으로 코딩되는 경우 비트스트림은 32 비트의 비트맵과 세 컬러 성분에 대한 두 개의 RV값으로 구성된다. 이때의 비트스트림 길이는 80(=32+3×2×8) 비트이다. 목표 비트스트림 길이에 맞추기 위해서는 코딩된 비트스트림의 길이가 64 비트보다 작거나 같아야 한다. 본 논문에서는 RV의 하위 3 비트를 잘라 목표 압축률을 맞추도록 한다. 이 경우 발생할 수 있는 최대 에러는 $7(=111_2)$이다. 에러를 감소시키기 위해 하위 3 비트에 대한 자름 연산을 수행하기 전 반올림(rounding) 연산을 수행하도록 한다. 예를 들어, $210_{10}(=11010010_2)$과 $213_{10}(=11010101_2)$은 각각 $208_{10}(=11010000_2)$과 $216_{10}(=11011000_2)$으로 반올림된다.

2×16 코딩 블록과 2×8 코딩 블록은 가로로 긴 형태이므로 수직으로 이웃하는 블록간의 공간적 연관성이 높다. 따라서 현재 블록의 RV는 보통 바로 윗 블록의 RV와 비슷한 값을 갖는다. 현재 블록의 RV가 바로 윗 블록의 RV와 비슷한 값을 갖는 경우, 제안하는 알고리즘은 현재 블록의 RV 대신 현재 블록의 RV와 윗 블록의 RV 간의 차이를 코딩한다. 이

표 3.1 RV 코딩 모드

RV coding mode	RVΔ	Coding	Bits for RVs in a block
RV same	0	Coding skip	0
RV coding mode 0	1	1 bit for sign and 1 bit for magnitude	12 bits (=3×2×2)
RV coding mode 1	2 or 3	1 bit for sign and 2 bits for magnitude	18 bits (=3×2×3)
VQ-BTC	otherwise	5 bits for magnitude	30 bits (=3×2×5)

때 코딩되는 비트의 수는 RV 간의 차이에 의해 결정된다. RV에 할당되는 비트 수를 의미하는 RV 코딩 모드는 헤더로 코딩된다. RV 코딩 모드를 결정하기 위해 모든 컬러 성분에 대한 현재 블록과 윗 블록의 RV 절대차 (absolute difference)를 계산한다. 절대차를 계산하기에 앞서 하위 3 비트에 대한 반올림과 자름 연산을 각각의 RV에 적용한다. 식 (3.1)과 같이 이 절대차 값들 중 가장 큰 값을 RVΔ로 결정한다.

$$\begin{aligned} &RV_{\Delta}=\max(|RV0_{\Delta}^{R}|,|RV1_{\Delta}^{R}|,|RV0_{\Delta}^{G}|,|RV1_{\Delta}^{G}|,|RV0_{\Delta}^{B}|,|RV1_{\Delta}^{B}|) \\ &where\ RVn_{\Delta}^{color}=r.t.\left(RVn_{cur}^{color}\right)-r.t\left(RVn_{up}^{color}\right),\ n=0,1 \end{aligned} \tag{3.1}$$

이때, 식 (3.1)에서 $r.t.(\cdot)$는 하위 3 비트에 대한 반올림과 자름 연산을 의미하고, RVn_{Δ}^{color}는 실제로 코딩될 RV 차이값을 의미한다.

표 3.1은 RVΔ 범위에 따른 네 개의 RV 코딩 모드를 보인다. RVΔ가 0인 경우 현재 블록의 RV는 코딩되지 않는다. 본 논문에서는 이 RV 코딩 모드를 'RV 동일(same)'이라 명명한다. 해당 RV 코딩 모드에서는 현재 블록의 RV가 윗 블록의 RV로 복원된다. RVΔ가 1인 경우 해당 블록은 'RV 코딩 모드 0'으로 코딩이 된다. 이 모드에서는 RV 차이가 부호를 위한 1 비트, 크기를 위한 1 비트 등 총 2 비트로 표현된다. 이때, 이미 하

위 3 비트가 잘려있기 때문에 크기 비트 ‘1’은 실제로 8을 의미한다. RV 코딩 모드 0에서는 RV에 할당되는 총 비트 수가 12(=컬러 성분의 수(3) × RV 수(2) × 비트 수(2))이다. RVΔ가 2 또는 3인 경우 ‘RV 코딩 모드 1’이 선택되며, 이 모드는 RV를 코딩하기 위해 18 비트를 이용한다. RVΔ가 3보다 큰 경우 세 컬러 성분에 대한 기존의 RV0와 RV1이 코딩된다. 이 때, RV의 총 수가 6이고 하위 3 비트가 잘렸으므로 RV 코딩에는 총 30 비트가 이용된다. 이 코딩 모드는 [26]과 동일한 방법으로 RV를 코딩하므로 ‘VQ-BTC 모드’라 명명한다.

RV0가 RV1과 동일한 경우 비트맵이 코딩될 필요가 없다. 이 코딩 모드는 ‘단일-컬러 코딩 모드’라 명명한다. 그러나 코딩 모드의 총 수가 5로 증가하므로 헤더 비트의 크기가 2에서 3으로 증가한다. 이 경우 모든 블록에서 RV 코딩 모드와는 상관없이 헤더 비트가 1씩 증가하므로 전체 비트스트림의 길이가 크게 증가한다. 헤더 비트 증가없이 단일-컬러 코딩 모드를 이용하기 위하여 본 논문은 단일-컬러 코딩 모드와 VQ-BTC 모드가 동일한 헤더를 이용한다. 이때, 단일-컬러 코딩 모드를 VQ-BTC 모드와 구분하기 위하여 하나의 RV가 아닌 두 개의 동일한 RV를 코딩한다. 따라서 15 비트 대신 30 비트가 RV를 위해 코딩된다. 디코더는 두 개의 RV를 비교하여 단일-컬러 코딩 모드임을 알 수 있다. 이 방식은 RV에 해당하는 비트를 감소시킬 수 없지만 비트맵에 해당하는 비트를 감소시킬 수 있다. 일반적으로 단일-컬러 코딩 모드가 복잡한 영역에서 선택될 확률이 매우 낮다. 두 개의 2×8 코딩 블록은 일반적으로 복잡한 영역에서 이용된다. 따라서 단일-컬러 코딩 모드는 2×8 코딩 블록에서 이용하지 않는다.

표 3.2 블록 크기와 RV 코딩 모드에 따른 비트스트림 길이

The number of coding block × coding block size	RV coding mode	Header bits	Bitstream length (CR)
1×2×16	RV same	001	35 bits (21.94)
	RV coding mode 0	010	47 bits (16.34)
	RV coding mode 1	011	53 bits (14.49)
	VQ-BTC (Normal)	000	64 bits (12.00)
	VQ-BTC (Single-color coding)	000	33 bits (23.27)
2×2×8	RV same	101	35 bits (21.94)
	RV coding mode 0	110	59 bits (13.02)
	RV coding mode 1	111	71 bits (10.82)
	VQ-BTC	100	95 bits (08.08)

표 3.2는 하나의 2×16 코딩 블록 또는 두 개의 2×8 코딩 블록이 이용되었을 때의 RV 코딩 모드, 헤더, 압축률을 보인다. 헤더의 크기는 코딩 블록 크기를 위한 1 비트, RV 코딩 모드를 위한 2 비트 등 총 3 비트로 고정이 된다. 네 번째 열은 기본 2×16 블록에 대한 비트스트림 길이와 압축률(compression ratio; CR)을 보인다. 예를 들어, 하나의 2×16 코딩 블록에서 RV 코딩 모드 1이 이용될 경우 비트스트림의 길이는 53 비트이고, 압축률은 14.49이다. 해당 비트스트림은 3 비트 헤더, 32 비트 비트맵, 18 비트 RV 등으로 구성된다. 두 개의 2×8 코딩 블록에서 VQ-BTC 모드가 이용될 경우 비트스트림은 3 비트 헤더, 32 비트 비트맵, 60 비트 RV 등으로 구성된다. 따라서 비트스트림의 길이는 95 비트이다. 9개의 코딩 모드 중 7개의 코딩 모드는 압축률이 12보다 크거나 같다. 반면에 나머지 2개의 코딩 모드(2×8 코딩 블록의 RV 코딩 모드 1과 VQ-BTC 모드)는 목표 압축률 12를 달성할 수 없다. 압축률이 낮은 2×8 코딩 블록은 화질 향상에 이용되는데, 해당 내용은 다음 장에서 다루도록 한다.

3.1.2 블록 크기 선택 방법

본 논문에서는 2×16 기본 블록을 코딩하기 위하여 하나의 2×16 코딩 블록 또는 두 개의 2×8 코딩 블록을 이용한다. 하나의 2×16 코딩 블록은 표 3.2에서 보인 바와 같이 압축률을 높이기 위해 이용한다. 두 개의 2×8 코딩 블록은 복잡한 블록을 코딩하는데 이용된다. 이 장에서는 블록의 복잡도에 따라 코딩 블록의 크기를 선택적으로 결정하는 방법을 설명하도록 한다. 또한 압축률 12를 보장하기 위한 방법을 제안하도록 한다.

블록의 복잡도는 두 개의 레벨(RV0, RV1)로 코딩된 블록에서의 절대 에러 합으로 결정이 된다. 각 픽셀의 절대 에러는 식 (3.2)와 같이 계산된다.

$$\Delta_{RV-pixel}(n) = |pixel(n) - RV1| \cdot bitmap(n) + |pixel(n) - RV0| \cdot \{1 - bitmap(n)\} \tag{3.2}$$

각 픽셀의 절대 에러가 계산되면 식 (3.3)과 같이 블록 내부 모든 픽셀의 절대 에러 합을 계산한다. 그런 다음 그 값이 미리 설정한 역치값보다 클 경우 그 블록을 복잡한 블록이라 판단한다.

$$Block\ cplx = \begin{cases} 1, & \sum_{\{R,G,B\}} \sum_{n=0}^{N-1} \Delta_{RV-pixel}(n) > th_{cplx} \\ 0, & otherwise \end{cases} \tag{3.3}$$

식 (3.3)에서 th_{cplx}는 실험적으로 결정되며, 본 논문에서는 이 값으로 384를 이용하였다.

표 3.2에서 보인 바와 같이 두 개의 코딩 모드는 비트스트림의 길이가 목표 길이인 64보다 길다. 목표 압축률을 보장하기 위하여 제안한 비

트-절약 방법에 의해 절약된 비트를 이용한다. 만약 생성된 비트스트림의 길이가 해당 블록에 할당된 목표 비트스트림 길이와 절약 비트의 합보다 큰 경우 코딩 블록의 크기는 무조건 2×16으로 설정된다. 참고로 2×16 코딩 블록을 이용했을 때의 비트스트림 길이는 항상 목표 비트스트림 길이보다 작거나 같다. 절약 비트는 해당 블록에 대한 코딩이 완료될 때마다 업데이트된다. 만약 코딩된 비트스트림의 길이가 목표 길이인 64 비트보다 작을 경우 절약 비트의 수는 증가한다. 절약 비트를 이용하게 되면 블록당, 라인당 비트-레이트가 변하게 된다. 이는 비트스트림에 대한 임의 접근을 방해한다. 본 논문에서는 각각의 라인에 대한 임의 접근을 위하여 한 라인에 대한 코딩이 완료되면 절약 비트를 0으로 초기화한다. 만약 절약 비트가 초기화되지 않으면 화질은 더 향상이 되겠지만 크기가 큰 내부 버퍼가 요구되고, 짧은 시간에 다량의 비트가 출력될 수 있다. 따라서 실시간 하드웨어 구현이 어려워진다.

본 논문의 블록 크기 선택 방법이 이용되면 현재 코딩 블록과 윗 코딩 블록의 크기가 다를 수 있다. 그림 3.1은 현재 코딩 블록과 윗 코딩 블록간의 네 가지 유형을 보인다. 경우 1은 현재 기본 블록과 윗 기본 블록이 모두 하나의 2×16 코딩 블록으로 코딩되는 경우를 보인다. 이 경우 윗 코딩 블록의 RV가 현재 코딩 블록의 RV를 코딩하는데 이용된다. 경우 2는 윗 기본 블록은 하나의 2×16 코딩 블록으로 코딩되고, 현재 기본 블록은 두 개의 2×8 코딩 블록으로 코딩되는 경우를 보인다. 경우 2에서는 두 개의 2×8 코딩 블록 모두가 윗 2×16 코딩 블록의 RV를 이용하여 RV를 코딩한다. 경우 3에서는 윗 기본 블록이 두 개의 2×8 코딩 블록으로 코딩되고, 현재 기본 블록은 하나의 2×16 코딩 블록으로 코딩된 경우

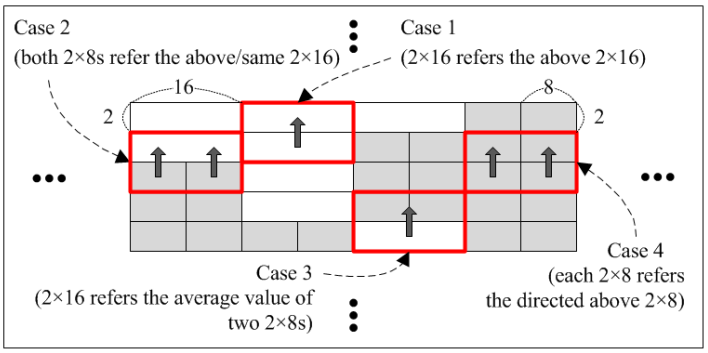


그림 3.1 코딩 블록 크기에 따른 윗 RV 참조 방법

를 보인다. 이 경우 현재 기본 블록은 RV가 한 쌍이지만 윗 기본 블록은 두 개의 2×8 코딩 블록이 이용되었으므로 RV가 두 쌍이다. 따라서 현재 코딩 블록은 윗 코딩 블록의 RV 평균값을 이용하여 RV를 코딩하도록 한다. 경우 4는 현재 기본 블록과 윗 기본 블록이 모두 두 개의 2×8 코딩 블록으로 코딩된 경우를 보인다. 이 경우, 각각의 2×8 코딩 블록은 수직으로 인접하는 2×8 코딩 블록의 RV를 이용하도록 한다.

3.1.3 알고리즘 요약

제안하는 BTC 알고리즘의 전체 순서도는 그림 3.2와 같다. 이때, 검은색으로 표시된 단계는 코딩 블록 크기를 결정하는 단계이고, 회색으로 표시된 단계는 코딩 모드를 결정하는 단계이다. 그리고 흰색 화살표와 회색 화살표는 각각 2×16 코딩 블록과 2×8 코딩 블록에 대한 실행 경로를 보인다. 제안 알고리즘의 첫 번째 단계는 하나의 2×16 코딩 블록과 두 개의 2×8 코딩 블록에 대한 RV와 비트맵을 계산하는 단계이다. 계산된 RV와 비트맵은 식 (3.2)와 식 (3.3)의 블록 복잡도 판단 계산에 이용된다. 코딩 블록 크기는 계산된 블록 복잡도 결과에 따라 결정된다. 각 RV의

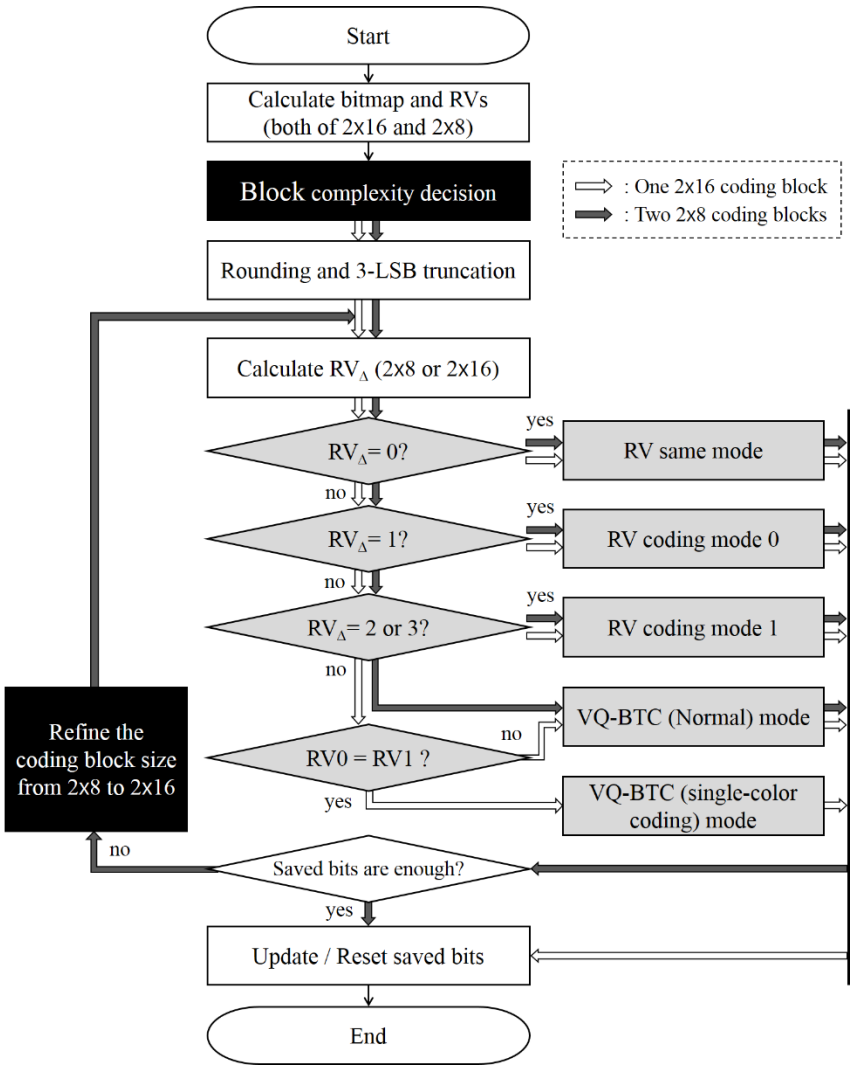


그림 3.2 제안하는 BTC 알고리즘의 전체 순서도

하위 3 비트에 대한 반올림 연산과 자름 연산을 수행한 후 식 (3.1)의 RVΔ에 따라 코딩 모드를 결정한다. 참고로 현재 기본 블록이 두 개의 2×8 코딩 블록으로 코딩되는 경우 두 개의 코딩 블록이 각각의 RV를 가지고 있으므로 식 (3.1)에서 총 12개의 절대차가 이용된다.

기본 블록이 하나의 2×16 코딩 블록으로 코딩이 되고, VQ-BTC 모드

가 선택된 경우 코딩 모드는 일반 모드 또는 단일-컬러 코딩 모드가 될 수 있다. 일반 모드에서는 코딩된 비트스트림의 길이가 65(=3+32+3×2×5) 비트이다. 따라서 목표 비트스트림의 길이인 64 비트를 맞출 수 없다. 이 문제를 해결하기 위하여 비트맵의 가운데에 위치한 1 비트가 코딩에 생략된다. 이미지가 복원될 때, 이 생략된 비트맵은 주변 픽셀의 비트맵을 이용하여 예측된다.

코딩 블록의 크기와 코딩 모드가 결정되면 비트스트림의 길이가 표 3.2와 같이 결정된다. 만약 코딩되는 비트스트림의 길이가 '절약 비트 수 + 64' 보다 작거나 같은 경우 해당 기본 블록은 결정된 코딩 블록 크기와 코딩 모드로 코딩된다. 그렇지 않을 경우 목표 압축률 12를 맞출 수 없기 때문에 코딩 블록 크기와 코딩 모드를 수정해야 한다. 이 경우 코딩 블록 크기는 2×16으로 결정되고, 2×16 코딩 블록에 대한 RVΔ가 재계산된다. 그런 다음, 2×16 코딩 블록에 대한 코딩 모드가 재선택된다. 해당 블록에 대한 코딩이 완료될 때, 최종 비트스트림이 생성되고, 절약 비트의 수가 업데이트된다. 한 라인에 대한 코딩이 완료되면, 절약 비트의 수가 0으로 초기화된다.

3.2 하드웨어 구조

이 장은 제안하는 알고리즘에 대한 실시간 처리 하드웨어 구조를 보이도록 한다. 제안하는 하드웨어 구조는 높은 처리량을 위하여 입력 데이터와 출력 데이터가 래스터 주사 순서로 매 사이클당 4 픽셀의 속도로 전송된다. 이를 위해서는 인코더와 디코더의 처리 속도가 매우 빨라야 한

다. 제안하는 하드웨어 구조에서는 압축된 데이터가 온-칩(on-chip) SRAM으로 구성된 프레임 메모리에 저장된다. 이때, 하드웨어 비용을 감소시키기 위하여 단일-포트 SRAM이 이용된다. 프레임 메모리 크기는 압축된 1-프레임 데이터 크기와 동일하며, 프레임 메모리의 대역폭은 매 사이클당 16 비트이다.

3.2.1 프레임 메모리 인터페이스

제안하는 알고리즘은 높이가 2인 블록을 코딩한다. 본 논문에서는 짝수 번째 라인의 입력 데이터가 들어올 때를 '짝수 라인 시간'이라 명명하고, 홀수 번째 라인의 입력 데이터가 들어올 때를 '홀수 라인 시간'이라 명명하도록 한다. 인코딩 동작은 두 라인의 데이터를 받아야 진행할 수 있다. 짝수 라인 시간에는 단순히 짝수 라인 데이터를 라인 버퍼에 저장한다. 그리고 홀수 라인 시간에 인코더가 인코딩을 수행하고, 생성된 비트스트림이 프레임 메모리로 전송된다. 따라서 인코더는 홀수 라인 시간에만 프레임 메모리에 접근한다. 프레임 메모리가 단일-포트 SRAM으로 구성되었기 때문에 프레임 메모리에 인코더와 디코더가 동시에 접근할 수 없다. 따라서 디코더는 비는 시간인 짝수 라인 시간에만 프레임 메모리에 접근하도록 한다. 이 시간 동안에 이전 프레임의 두 라인에 해당하는 비트스트림이 디코더에 전달된다.

그림 3.3 (a)는 인코딩 과정의 프레임 메모리 인터페이스를 보인다. 입력 데이터는 매 사이클당 4 픽셀의 속도로 인코더에 전달된다. 짝수 라인의 데이터는 잠시 인코더 라인 버퍼에 저장이 된다. 그리고 홀수 라인의 데이터가 들어오는 홀수 라인 시간에 짝수 라인의 데이터와 홀수 라

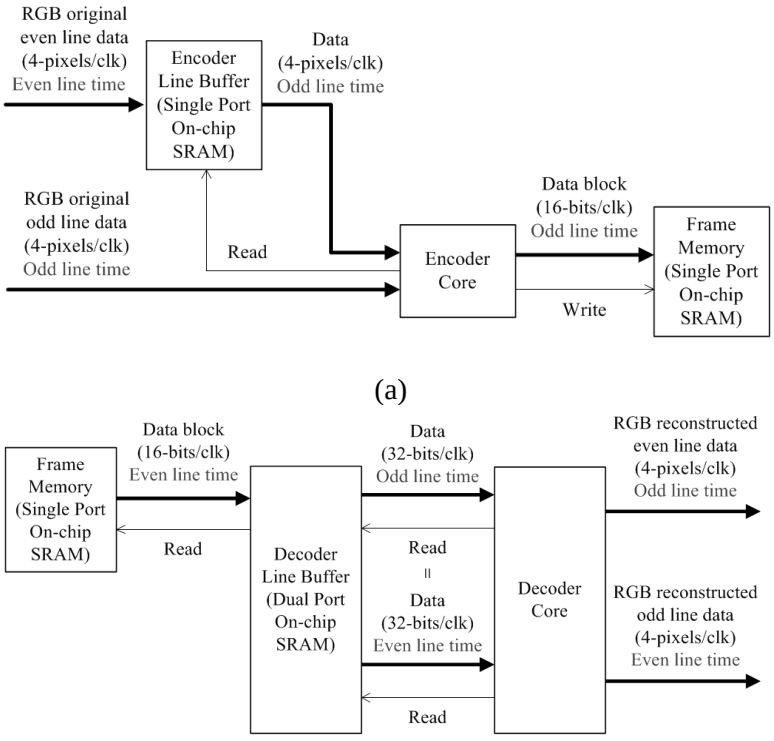


그림 3.3 프레임 메모리 인터페이스. (a) 인코더 (b) 디코더

인의 데이터가 인코더 코어에 전달이 된다. 이때, 짝수 라인의 데이터는 라인 버퍼로부터 전달되고, 홀수 라인의 데이터는 입력 데이터가 직접 전달이 된다. 인코더 코어는 2×16 코딩 블록을 4 사이클 동안 처리하기 위하여 매 사이클마다 192 비트를 전달받는다. 기본 블록에 대한 데이터가 모두 도착하면 인코더 코어가 인코딩을 시작하고, 생성된 비트스트림은 프레임 메모리로 전송된다.

제안한 알고리즘이 생성하는 비트스트림의 길이는 33 비트에서부터 95 비트까지이다. 본 논문에서는 프레임 메모리 비용을 감소시키기 위하여 메모리의 대역폭을 사이클당 16 비트로 설정하였다. 이와 같은 조건에

서는 4 사이클 동안 최대 64 비트가 전송될 수 있다. 만약 선택된 코딩 모드가 64 비트보다 더 긴 비트스트림을 생성할 경우 비트스트림의 일부는 프레임 메모리로 출력되지 못한다. 따라서 인코더는 출력되지 못한 비트스트림을 저장하기 위하여 내부 메모리를 이용하는데, 그 크기에 대해서는 다음 장에서 설명하도록 한다.

디코딩 과정의 프레임 메모리 인터페이스는 그림 3.3 (b)와 같다. 디코더는 두 라인에 해당하는 비트스트림을 프레임 메모리로부터 읽어오고, 해당 비트스트림은 짝수 라인 시간 동안 디코더 라인 버퍼에 저장된다. 두 라인에 대한 데이터가 준비되면 디코더는 디코딩을 시작한다. 이미지는 블록 단위, 그리고 좌측에서 우측 순서로 복원된다. 하지만 일반적인 LCD 오버드라이브 유닛은 래스터 주사 순서로 데이터를 전달받는다. 다시 말해, 짝수 번째 라인의 복원된 데이터가 모두 전송된 다음, 홀수 번째 라인의 복원된 데이터가 전송될 수 있다. 이 문제를 해결하기 위한 가장 간단한 방법은 복원된 데이터를 저장하는 버퍼를 이용하는 방식이다. 이 경우 홀수 라인 시간 동안에는 복원된 짝수 라인 데이터가 오버드라이브 유닛으로 전송이 되고, 홀수 라인 데이터는 버퍼에 저장이 된다. 그리고 짝수 라인 시간 동안에는 저장되었던 홀수 라인 데이터가 오버드라이브 유닛으로 전송이 된다. 이 방법은 한 라인의 복원된 데이터를 저장하기 위한 추가 버퍼를 필요로 한다. 이와 같은 버퍼는 하드웨어 비용에 큰 영향을 미치게 된다. 따라서 본 논문은 버퍼를 쓰는 방법 대신에 동일한 디코딩을 두 번 수행하는 방식을 이용하도록 한다. 홀수 라인 시간의 경우 복원된 두 라인의 데이터 중 짝수 번째 라인의 데이터만 오버드라이브 유닛으로 전달이 된다. 즉, 복원된 홀수 번째 라인의 데이터는 버려

진다. 반면에 짝수 라인 시간 동안에는 홀수 라인 시간에 수행하였던 동일한 동작이 반복되며, 복원된 두 라인 중 홀수 번째 라인에 해당하는 데이터가 전송된다. 이때, 디코딩과 비트스트림을 읽어오는 과정이 동시에 수행되므로 디코더 라인 버퍼는 이중-포트 SRAM으로 구성된다.

복원된 픽셀을 매 사이클당 4 픽셀의 속도로 전송하기 위해서는 모든 코딩 모드에 대한 비트스트림이 4 사이클 이내에 읽어져야 한다. 하지만 코딩 모드 중 2개는 비트스트림의 길이가 64 비트를 초과하고, 해당 비트스트림은 16 비트 대역폭에서 4 사이클 내에 전송될 수 없다. 따라서 디코더 라인 버퍼의 대역폭은 매 사이클당 32 비트로 설정이 된다. 이를 위해 디코더 라인 버퍼는 두 개의 16 비트 데이터를 프레임 메모리로부터 전달받아 연결시킨 후 32 비트 단위로 라인 버퍼에 저장한다.

3.2.2 인코더와 디코더의 구조

제안하는 인코더와 디코더 하드웨어 구조는 그림 3.4와 같다. 회색으로 표시된 사각형은 온-칩 SRAM으로 구현된 모듈을 의미한다. 인코더 구조의 경우 2×16 기본 블록에 해당하는 픽셀 데이터가 4 사이클 동안 전송된다. 따라서 모든 모듈의 동작은 실시간 동작을 위하여 4 사이클 이내에 완료되어야 한다. 제안하는 인코딩 파이프라인 구조는 그림 3.5에서 확인할 수 있다. 모든 모듈은 2×16 기본 블록을 4 사이클 동안에 처리한다. 디코더의 경우 헤더에 따라 코딩 블록 크기와 RV 코딩 모드를 결정하고, 파싱(parsing)된 RV와 비트맵을 이용하여 간단히 픽셀을 복원한다. 따라서 디코딩 파이프라인 구조는 인코더 파이프라인 구조에 비해 매우 단순하다.

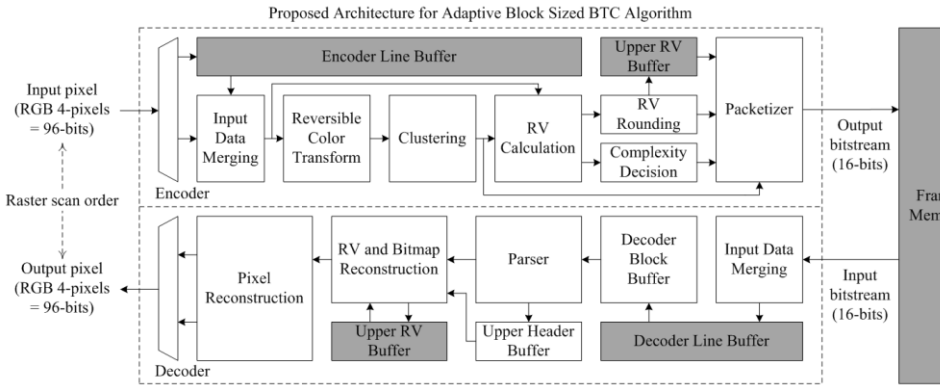


그림 3.4 제안하는 인코더와 디코더 구조

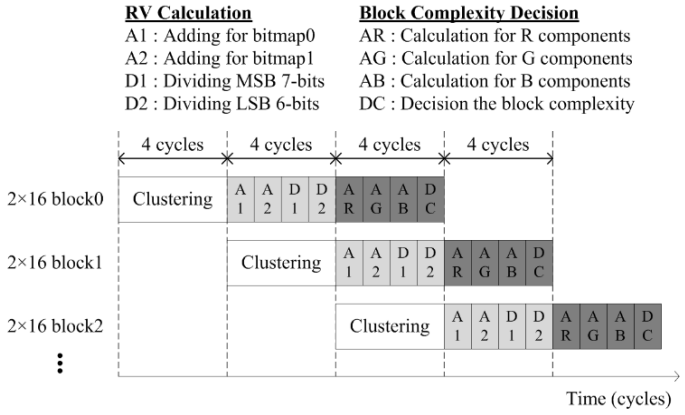


그림 3.5 인코딩 파이프라인 구조

그림 3.4에서 확인할 수 있듯이 입력 데이터 결합(merging) 모듈에서 모인 짝수 라인 데이터와 홀수 라인 데이터는 가역(reversible) 컬러 변환 모듈로 전달된다. 컬러 이미지 압축을 위한 BTC 알고리즘에서는 블록 내부의 픽셀을 클러스터링시키는 방법이 복원 이미지의 화질에 매우 큰 영향을 미친다. 기존 알고리즘은 K-means 클러스터링이나 PCA와 같은 방법을 이용하였다. 그러나 이와 같은 방법들은 실시간 하드웨어 구현이 매우

어려운 문제가 있다. 본 논문에서는 클러스터링 방식을 단순화하기 위하여 RGB 컬러 성분을 Y 성분으로 먼저 변환하도록 한다. Y 성분은 JPEG2000에서 이용하는 (3.4)로 계산할 수 있다[2].

$$\mathrm{Y} = \frac{R + 2G + B}{4} \tag{3.4}$$

Y 성분이 계산되면 블록 내부에 존재하는 최대 Y값과 최소 Y값을 구하고 그 값들의 평균을 계산해 클러스터링의 역치값으로 이용한다. 최대 Y값과 최소 Y값의 평균값은 클러스터링의 역치값에 튀는 Y값을 많이 반영시킬 수 있기 때문에 모든 Y값들의 평균값보다 가장자리 정보를 보존하는데 더 유리하다. 이와 같은 클러스터링 방식을 이용하여 클러스터링 모듈은 RV 생성, 블록 복잡도 계산, 실제 비트스트림 포함 등에 이용되는 비트맵을 생성한다.

그림 3.6은 인코더에서 가장 많은 연산량을 보유한 RV 계산 모듈의 구조를 보인다. 해당 모듈의 경우 아직 코딩 블록의 크기와 코딩 모드가 결정되지 않은 시점이기 때문에 모든 코딩 블록에 대한 RV 값을 계산한다. 그림 3.6은 2×16 코딩 블록의 R 성분에 대한 회로만을 보인다. 입력 RGB 데이터는 그림 3.6에서 시프트 레지스터로 표기된 내부 버퍼에 먼저 저장이 된다. 이는 입력 RGB 데이터가 비트맵에 비해 먼저 도착하기 때문이다. RV 계산 모듈은 총 네 개의 하위 단계를 가지고 있다. 첫 번째 사이클에는 비트맵0에 해당하는 픽셀들이 모두 더해진다(A1). 그리고 다음 사이클에는 비트맵1에 해당하는 픽셀들이 동일한 덧셈기 트리를 이용하여 더해진다(A2). 피제수와 제수의 비트-너비는 각각 13 비트와 6 비트이다. 일반적으로 나눗셈은 긴 임계 경로를 갖기 때문에 본 논문에서는 남은 두 사이클을 이용하여 나눗셈을 수행한다. 피제수의 상위 7 비트의

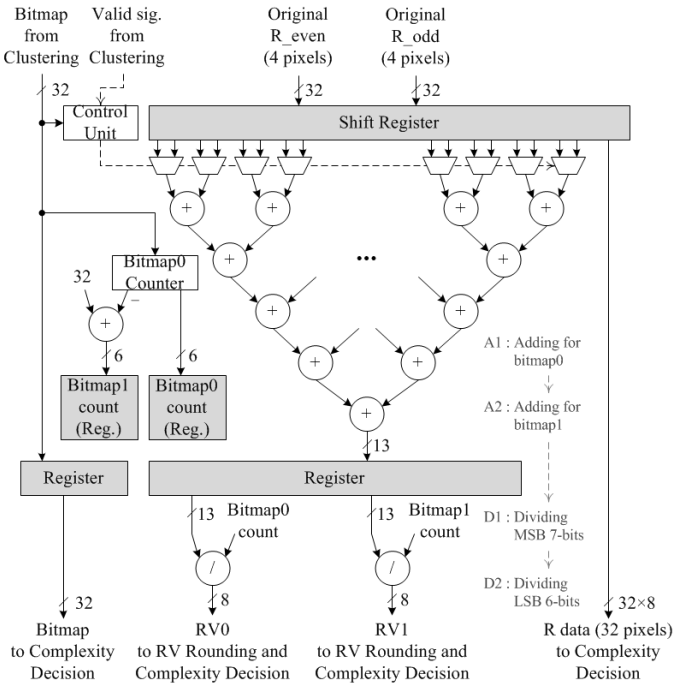


그림 3.6 RV 계산 모듈의 구조

경우 세 번째 사이클에 처리가 되고(D1), 하위 6 비트의 경우 네 번째 사이클에 처리가 된다(D2).

RV 반올림 모듈은 목표 압축률을 맞추기 위하여 RV의 하위 3 비트를 반올림하고 잘라낸다. 그림 3.7의 블록 복잡도 판단 모듈은 인코더에서 두 번째로 많은 연산량을 보이는 회로이다. 이 모듈은 RGB 입력 데이터, RV, 비트맵 등을 이전 모듈들로부터 전달받는다. 그런 다음, 식 (3.2)와 식 (3.3)을 통해 현재 블록의 복잡도를 판단한다. 이때, 하드웨어 이용 효율을 높이기 위하여 세 개의 컬러 성분이 하나의 덧셈기 트리를 세 사이클 동안 공유한다(AR, AG, and AB). 해당 시간 동안 전체 합이 저장 및 업데이트된다. 마지막 네 번째 사이클에는 전체 합이 임계값과 비교되고, 최종 블록의 복잡도가 결정된다(DC).

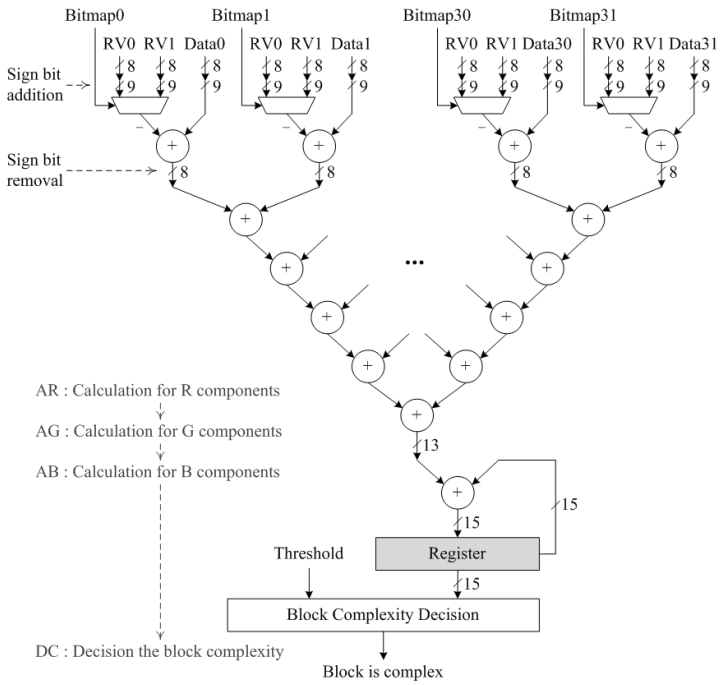


그림 3.7 블록 복잡도 판단 모듈의 구조

기본 블록에 대한 RV, 비트맵, 블록의 복잡도 등은 패킷타이저(packetizer) 모듈로 전달이 된다. 해당 모듈은 블록의 복잡도와 절약 비트를 기반으로 코딩 블록 크기를 결정한다. 코딩 모드는 현재 블록과 윗 블록의 RV 비교를 통해 결정이 된다. 윗 블록의 RV를 저장 및 이용하기 위해 윗 RV 버퍼가 이용된다. 최종적으로 생성된 비트스트림은 매 사이클당 16 비트의 속도로 프레임 메모리에 전송이 된다.

표 3.2에서 보인 것처럼 제안하는 구조는 총 9개의 코딩 모드를 이용하며, 그 중 2개의 모드(두 2×8 코딩 블록의 RV 코딩 모드 1과 VQ-BTC 모드)는 목표 길이인 64 비트를 초과하는 비트스트림을 생성한다. 본 논문에서는 프레임 메모리의 대역폭이 매 사이클당 16비트로 제한되었으므로 해당 두 모드의 7 비트 또는 31 비트가 네 사이클 이내에 전송될 수

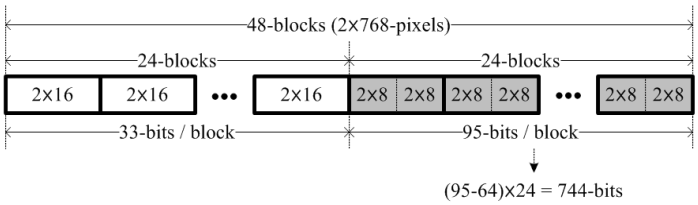


그림 3.8 패킷타이저 모듈에서의 최악의 내부 버퍼 크기

없다. 이와 같은 문제를 해결하기 위하여 패킷타이저 모듈은 내부 버퍼를 이용한다. 내부 버퍼의 크기는 프레임 너비와 코딩 모드의 종류에 의해 결정된다. 그림 3.8은 가장 많은 비트가 전송되지 못하는 최악의 경우를 보인다. 이때, 프레임 너비는 768이다. 그림 3.8에서 굵은 사각형은 2×16 기본 블록을 의미한다. 프레임의 너비와 기본 블록의 크기가 각각 768과 2×16이므로 한 라인에는 총 48개의 블록이 존재한다. 최악의 경우는 좌측 24개 블록이 2×16 코딩 블록의 단일-컬러 코딩 모드로 코딩이 되고, 우측 24개 블록이 두 2×8 코딩 블록의 VQ-BTC 모드로 코딩되는 경우이다. 이 경우 우측 24개 블록에서는 4 사이클마다 31 비트씩 내부 버퍼에 비트가 누적이 된다. 따라서 요구되는 내부 버퍼의 크기는 744 비트이다. 하지만 최악의 경우가 발생할 확률이 매우 낮기 때문에 744 비트 크기의 내부 버퍼는 그 이용 효율이 매우 낮다.

본 논문에서는 이 내부 버퍼의 크기를 실험적으로 결정하도록 한다. 그림 3.9는 내부 버퍼 크기에 따른 화질 변화를 보인다. 가로축과 세로축은 실험 이미지와 PSNR(peak signal-to-noise ratio)을 의미한다. 그림 3.9에서 확인할 수 있듯이 내부 버퍼 크기가 256 비트를 넘어가면 화질은 포화상태(saturation)가 된다. 따라서 본 논문은 내부 버퍼 크기를 256 비트로 설정하였다. 설정된 내부 버퍼 크기가 최악의 경우의 버퍼 크기보다

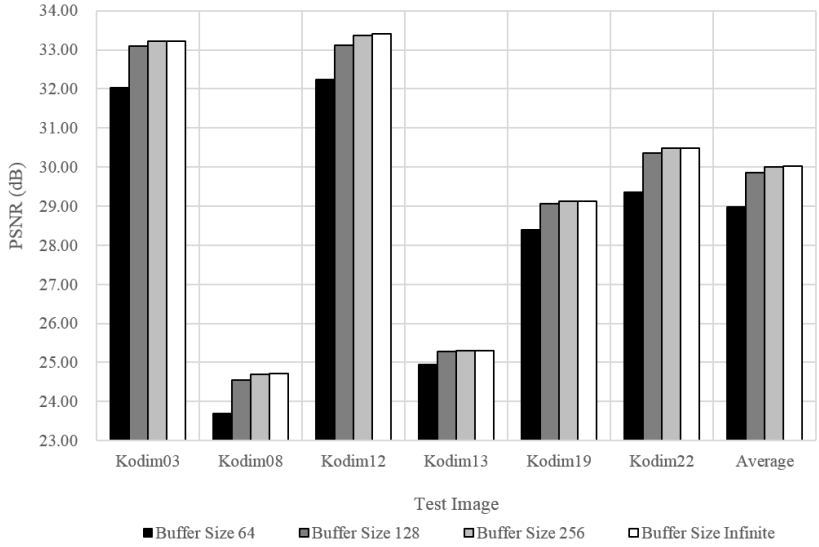


그림 3.9 패킷타이저 모듈의 내부 버퍼 크기에 따른 PSNR 변화

작기 때문에 내부 버퍼 크기가 오버플로(overflow)되는 상황을 방지하기 위한 방법이 필요하다. 본 논문에서는 내부 버퍼의 오버플로가 발생하는 경우 코딩 블록 크기가 2×16으로 강제 전환이 된다. 2×16 코딩 블록의 경우 비트스트림의 길이가 항상 64 비트보다 작거나 같기 때문에 내부 버퍼에 대한 오버플로가 발생할 수 없다. 해당 방식의 경우 코딩 블록 크기와 코딩 모드를 변화시키므로 약간의 화질 감소가 동반된다.

제안하는 디코더 하드웨어의 경우, 짝수 라인 시간에 프레임 메모리로부터 매 사이클당 16 비트씩 비트스트림을 전달받는다. 그런 다음, 입력 데이터 결합 모듈이 두 개의 16 비트를 모아 32 비트 단위로 디코더 라인 버퍼에 저장한다. 홀수 라인 시간 동안 2×16 기본 블록에 해당하는 비트스트림이 디코더 블록 버퍼에 전달된다. 전달된 비트스트림은 파서(parser) 모듈에서 헤더, RV, 비트맵 등으로 파싱된다. 파싱된 헤더 정보를 이용하여 코딩 블록 크기와 코딩 모드가 결정된다. 코딩 블록 크기를 의

미하는 1 비트 크기의 헤더는 다음 라인 디코딩을 위해 윗 헤더 버퍼에 저장이 된다. 파싱된 RV와 비트맵은 RV-비트맵 복원 모듈에서 복원이 된다. 그리고 복원된 RV는 다음 라인 디코딩을 위해 윗 RV 버퍼에 저장된다. 최종 출력 픽셀은 픽셀 복원 모듈에서 복원된다. 복원된 픽셀은 매 사이클당 4 픽셀의 속도로 오버드라이브 유닛에 전송된다. 제안한 디코더 하드웨어 구조는 동일한 디코딩 동작을 두 번 반복한다. 그러나 두 번의 동작이 다른 시간대에 이루어지므로 디코더는 하나만 존재한다.

3.3 실험 결과

이 장은 제안한 알고리즘과 이전 알고리즘의 성능을 비교하도록 한다. 또한 제안 알고리즘에 대한 하드웨어 구현 결과를 보이도록 한다.

3.3.1 알고리즘 성능

표 3.3은 기존 알고리즘과 제안 알고리즘의 PSNR 성능을 보인다. 실험을 위해 768×512 또는 512×768 크기의 Kodak 컬러 이미지가 이용되었다[41]. 표 3.3에는 대표적인 6개의 Kodak 이미지와 24개의 Kodak 이미지 평균 결과가 표시되어 있다. 이와 더불어 고해상도에서의 성능을 보이기 위해 4K UHD(Traffic), full HD(Kimono, Park scene, and Basketball drive), HD(Johnny, Kristen and Sara, and Vidyo1) 시퀀스의 첫 번째 프레임도 이용되었다. 제안한 알고리즘은 고정 압축률을 목표로 하지만 실제 압축률(measured CR)은 고정 압축률보다 클 수 있다. 따라서 소괄호에 실제 압축률도 함께 표기하였다.

표 3.3 이전 방식과 제안 방식의 PSNR(DB) 성능 비교

	PSNR (measured CR)						
Algorithm	Conv. BTC [3]	VQ-BTC [27]	AM-BTC [28]		Proposed		
Block size	4×4	4×4	4×4	2×16	2×4	2×6	2×16
Target CR	4.00	6.00	6.00	12.00	4.68	6.40	12.00
Kodak03	37.26	35.37	36.58	32.06	37.40 (7.39)	36.04 (9.67)	33.29 (15.22)
Kodak08	27.86	26.89	29.85	25.03	30.82 (4.80)	28.39 (6.55)	24.60 (12.16)
Kodak12	36.74	34.97	36.47	32.68	37.51 (6.99)	35.76 (9.36)	33.49 (14.82)
Kodak13	27.71	27.16	29.79	24.69	28.85 (4.78)	27.55 (6.53)	25.27 (12.16)
Kodak19	31.78	31.16	34.66	29.84	35.07 (5.92)	34.02 (7.62)	29.13 (13.79)
Kodak22	33.65	32.15	34.36	29.38	35.27 (5.79)	33.53 (7.87)	30.47 (13.41)
Average	33.26	31.92	34.27	29.36	34.71 (6.15)	33.10 (8.11)	30.03 (13.69)
Traffic	33.78	N/A	35.37	30.96	37.81 (5.61)	35.25 (7.55)	31.14 (13.16)
Kimono	39.79		39.21	33.82	39.49 (6.05)	38.11 (8.21)	34.85 (13.87)
Park scene	34.82		35.34	31.10	37.30 (5.06)	35.14 (6.98)	31.84 (12.69)
Basketball drive	36.45		37.07	33.30	38.85 (5.70)	37.35 (8.15)	34.59 (14.57)
Johnny	36.52		38.48	32.97	38.80 (7.00)	37.10 (9.29)	33.44 (14.95)
Kristen and Sara	34.52		37.20	31.79	38.24 (7.07)	36.16 (9.34)	32.15 (14.77)
Vidyo1	36.22		38.68	33.27	38.51 (6.71)	36.49 (9.13)	33.57 (14.41)
Average	36.01	N/A	37.34	32.46	38.43 (6.17)	36.51 (8.38)	33.08 (14.06)

보편적인 BTC[3]는 4×4 블록을 코딩하는데, R, G, B 컬러 성분이 따로 코딩되므로 압축률이 4이다. VQ-BTC[27] 역시 4×4 블록을 코딩하지만

R, G, B 컬러 성분이 단일 비트맵을 공유한다. 따라서 압축률이 6으로 증가한다. AM-BTC[28]의 압축률은 VQ-BTC와 마찬가지로 6이지만 2-레벨과 4-레벨이 선택적으로 이용되므로 전반적인 이미지 화질이 향상된다. 본 논문에서는 공정한 비교를 위하여 기존 AM-BTC를 2×16 블록 크기로 수정하였고, 그 때의 목표 압축률은 12이다. 이를 위하여 RV의 하위 비트를 잘랐고, 4-레벨의 비트맵은 서브-샘플링되었다. 참고로 서브-샘플링이 없는 4-레벨의 비트맵은 비트맵만으로 목표 비트스트림 길이인 64 비트를 채운다. 따라서 목표 압축률을 맞추기 위하여 4-레벨의 비트맵을 서브-샘플링하였다. 제안 방식을 기존 방식과 조금 더 대등한 위치에서 비교하기 위하여 다양한 압축률에 대한 결과를 함께 보였다. 이를 위해 기본 블록의 크기를 조절하였다. 표 3.3의 여섯 번째 열은 목표 압축률이 4.68인 경우의 실험 결과를 보인다. 이 실험에서는 하나의 2×4 코딩 블록 또는 두 개의 2×2 코딩 블록이 선택적으로 이용되며, th_{cplx}는 64로 조정되었다. 표 3.3의 일곱 번째 열은 목표 압축률이 6.40인 경우의 실험 결과를 보인다. 이 경우 하나의 2×6 코딩 블록 또는 두 개의 2×3 코딩 블록이 선택적으로 이용되며, 128이 th_{cplx} 값으로 이용된다. 표 3.3의 두 번째 열과 여섯 번째 열의 비교를 통해 보편적인 BTC 방식에 비해 제안 방식의 목표 압축률이 0.68 더 높음에도 불구하고 1.45 dB 더 높은 평균 PSNR을 보이는 것을 확인할 수 있다. 표 3.3의 세 번째 열과 일곱 번째 열의 비교를 통해 제안 방식이 VQ-BTC보다 0.40 더 높은 목표 압축률을 가짐에도 1.18 dB 더 높은 평균 PSNR을 보이는 것을 확인할 수 있다. 네 번째 열과 일곱 번째 열의 비교를 통해 제안 방식의 평균 PSNR이 4×4 블록을 코딩하는 AM-BTC의 평균 PSNR보다 더 낮은 수치를 보이는 것을 확인

할 수 있다. 하지만 제안 방식의 목표 압축률은 4×4 블록을 코딩하는 AM-BTC보다 더 높고, 이용하는 코딩 블록의 높이가 더 짧다. 2×16 블록을 코딩하고 목표 압축률이 12인 AM-BTC와 비교할 경우 제안 방식의 평균 PSNR이 0.67 dB만큼 더 높다. 4K UHD, Full HD, HD 크기의 이미지에서는 제안 방식이 2×16 블록을 코딩하는 AM-BTC보다 0.62 dB만큼 더 높은 평균 PSNR을 보인다.

그림 3.10은 이전 방식과 제안 방식에 의해 복원된 이미지를 보인다. 그림 3.10 (a)는 가장자리 정보를 많이 포함하고 있는 Kodak19 원본 이미지의 한 부분을 보인다. 그림 3.10 (b), (c), (d), (e), (f)는 각각 보편적인 BTC, VQ-BTC, 4×4 블록을 코딩하는 AM-BTC, 2×16 블록을 코딩하는 AM-BTC, 제안 방식의 복원 이미지를 보인다. 목표 압축률이 12임을 감안할 때 제안 방식은 가장자리 정보를 상당히 잘 보존한다. 2×16 블록을 코딩하는 AM-BTC 역시 괜찮은 화질을 보이지만 4-레벨에서 이용되는 하나의 기준값에 의한 컬러 변화, 비트맵 서브-샘플링에 의한 부자연스러운 현상 등이 관찰된다.

표 3.4는 제안한 비트-절약 방법을 이용했을 때의 선택된 RV 코딩 모드의 비율을 보인다. 해당 비율을 측정하기 위해 24개의 Kodak 컬러 이미지가 이용되었고, 평균 비율이 표 3.4에 표기되었다. 하나의 2×16 코딩 블록이 이용되는 경우 높은 압축률을 갖는 코딩 모드들이 상대적으로 많이 선택된다. 두 개의 2×8 코딩 블록이 이용되는 경우 상대적으로 낮은 압축률을 갖는 코딩 모드들이 많이 선택된다. 이는 제안한 비트-절약 방식이 낮은 복잡도를 갖는 부분에서 높은 효율을 보임을 의미한다.

표 3.5는 제안한 블록 크기 선택 방식을 이용했을 때 선택된 코딩 블

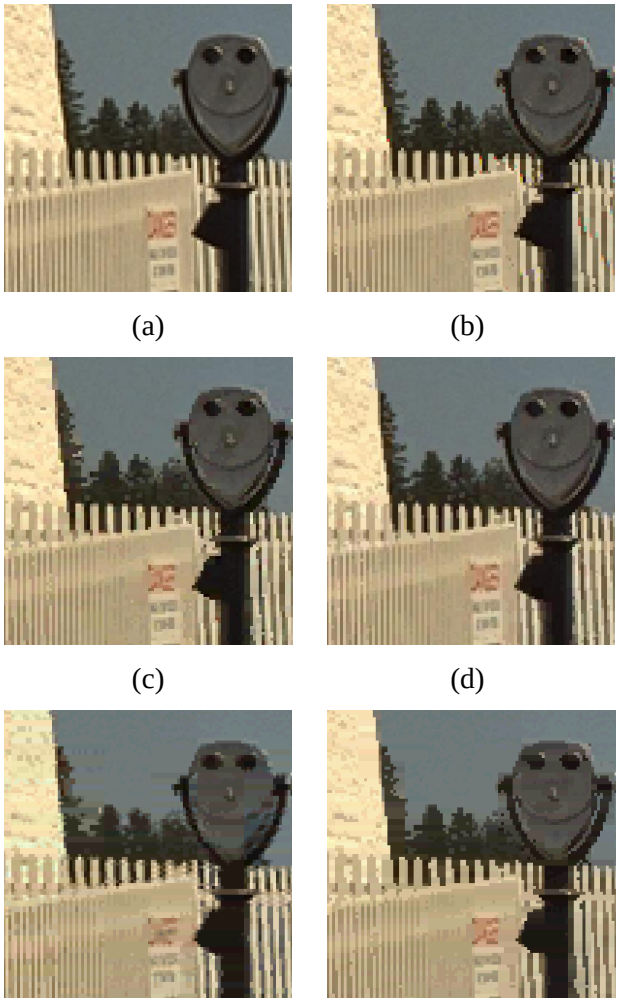


그림 3.10 Kodak19 (365, 384) ~ (464, 483) 이미지. (a) 원본 이미지 (b) 보편적인 BTC[3]의 복원 이미지 (c) VQ-BTC[27]의 복원 이미지 (d) 4×4 블록을 코딩하는 AM-BTC[28]의 복원 이미지 (e) 2×16 블록을 코딩하는 AM-BTC[28]의 복원 이미지 (f) 제안 방식의 복원 이미지

록 크기의 비율과 그 때의 평균 PSNR 결과를 보인다. 이전 실험과 마찬가지로 24개의 Kodak 컬러 이미지가 이용되었다. th_{cplx}가 무한대인 경우

표 3.4 선택된 RV 코딩 모드의 비율

Coding block size	Coding mode	CR	Average ratio (%)
One 2×16 coding block	RV same	21.94	14.95
	RV coding mode0	16.34	31.11
	RV coding mode 1	14.49	10.31
	VQ-BTC (Normal)	12.00	7.74
	VQ-BTC (single-color coding)	23.27	0.29
Two 2×8 coding blocks	RV same	21.94	0.30
	RV coding mode 0	13.02	9.66
	RV coding mode 1	10.82	17.14
	VQ-BTC	8.08	8.49

표 3.5 선택된 코딩 블록 크기의 비율과 평균 PSNR

th_{cplx}	Average ratio (%)		Average PSNR (dB)
	One 2×16	Two 2×8s	
Infinite	100.00	0.00	28.97
512	72.24	27.76	30.02
384	64.41	35.59	30.03

모든 2×16 기본 블록은 하나의 2×16 코딩 블록으로 코딩이 된다. 이 경우 평균 PSNR은 28.97 dB이다. th_{cplx}가 512인 경우 두 개의 2×8 코딩 블록이 27.76%만큼 이용되었고, 평균 PSNR은 30.02 dB이다. th_{cplx}가 384인 경우 두 개의 2×8 코딩 블록은 35.59%만큼 이용되고, 평균 PSNR은 30.03 dB로 향상된다. 이와 같은 결과는 제안한 블록 크기 선택 방법이 복잡한 영역의 화질을 효과적으로 향상시키고 있음을 의미한다.

3.3.2 하드웨어 구현 결과

이 장은 제안한 알고리즘에 대한 하드웨어 구현 결과를 보인다. 표 3.6에서 표 3.8까지는 ASIC 0.13μm에 대한 합성 결과를 보인다. 제안 하

표 3.6 제안 하드웨어의 처리 속도 (ASIC 0.13 μm)

Module	Maximum frequency (MHz)	Throughput (Gbps)	
		w/ frame memory interface (overall system)	w/o frame memory interface (only coder)
Encoder	143	13.7	27.5
Decoder	333	32.0	63.9

표 3.7 제안 하드웨어의 메모리 크기 (ASIC 0.13 μm)

Memory		Bits
Encoder	Total	21,312
	Encoder Line Buffer	18,432
	Upper RV Buffer	2,880
Decoder	Total	5,952
	Decoder Line Buffer	3,072
	Upper RV Buffer	2,880

드웨어 구조의 처리 속도는 표 3.6에서 확인할 수 있다. 구현한 인코더와 디코더의 동작 주파수는 각각 143 MHz와 333 MHz이다. 3.2.1장에서 제안한 프레임 메모리 인터페이스에서는 RGB 데이터가 매 사이클당 96 비트(4 픽셀)씩 전송이 된다. 따라서 인코더, 디코더 하드웨어의 처리 속도는 각각 13.7 Gbps와 32.0 Gbps이다. 인코더, 디코더 코어 하드웨어는 2×16 기본 블록을 4 사이클 동안 처리한다. 인코더는 홀수 라인 시간에만 동작하고, 디코더는 동일한 동작을 짝수 라인 시간과 홀수 라인 시간에 반복한다. 따라서 표 3.6의 마지막 열에서 보인 것처럼 인코더, 디코더 하드웨어의 자체 처리 속도는 27.5 Gbps와 63.9 Gbps이다.

표 3.7은 제안 하드웨어가 이용하는 메모리 크기를 보인다. 이때, 메모리 크기는 압축하는 이미지의 너비가 768인 경우의 결과이고, 프레임 메모리는 제외하였다. 각 메모리는 온-칩 SRAM으로 구현되었다. 단, 패

킷타이저 모듈에서 이용되는 내부 버퍼의 경우 레지스터로 구현되었다. 인코더에서는 그림 3.4의 라인 버퍼와 윗 RV 버퍼가 메모리로 구현되었다. 인코더의 라인 버퍼는 한 라인에 해당하는 원본 데이터를 저장하며, 그 크기는 18,432(=768×3×8) 비트이다. 인코더의 윗 RV 버퍼는 한 라인에 해당하는 RV들을 저장하는데, 제안하는 방식의 경우 하위 3 비트를 잘라내기 때문에 총 크기가 2,880 비트이다. 인코더와 마찬가지로 디코더도 라인 버퍼와 윗 RV 버퍼를 메모리로 구현한다. 디코더의 윗 RV 버퍼의 경우 인코더와 그 크기가 동일하다. 그러나 디코더의 라인 버퍼는 1/12로 압축된 두 라인의 비트스트림을 저장하므로 그 크기가 3,072 비트이다.

표 3.8은 제안 하드웨어 구조의 게이트 수 결과를 보인다. 인코더의 전체 게이트 수는 68.7K이다. 이 중 RV 계산 모듈이 33.7K를 이용하여 전체 인코더의 49.1%를 차지하고 있다. RV 계산 모듈은 각 컬러 성분과 블록 크기에 대한 RV를 모두 계산하므로 많은 수의 덧셈기와 나눗셈기를 이용한다. 블록 복잡도 판단 모듈의 경우 14.2K 게이트를 이용하며, 이는 전체 인코더의 20.7%를 차지한다. 해당 모듈은 그림 3.7에서 보인 바와 같이 많은 수의 덧셈기를 이용한다. 패킷타이저 모듈의 경우 13.7K 게이트를 이용하고, 이는 전체 인코더의 19.9%를 차지한다. 이는 레지스터로 구현된 256 비트 크기의 내부 버퍼가 포함된 결과이다. 해당 모듈은 가능한 모든 경우의 수를 비교하고, 코딩 블록 크기와 RV 코딩 모드를 최종적으로 결정한다. 따라서 상대적으로 많은 게이트를 이용한다.

디코더의 전체 게이트 수는 13.6K로 인코더의 19.8%에 해당한다. 디코더는 헤더에 따라 RV, 비트맵 등을 간단히 파싱하고, 최종 픽셀을 복원

표 3.8 제안 하드웨어의 게이트 수 (ASIC 0.13 μm)

Module		Gate counts (Ratio)
Encoder	Total	68.7K (100%)
	RCT	1.0K (1.5%)
	Clustering	4.9K (7.1%)
	RV Calculation	33.7K (49.1%)
	Block Complexity Decision	14.2K (20.7%)
	RV Rounding	1.2K (1.7%)
	Packetizer	13.7K (19.9%)
Decoder	Total	13.6K (100.0%)
	Decoder Block Buffer	4.8K (35.3%)
	Parser	1.1K (8.1%)
	Upper Header Buffer	0.5K (3.7%)
	RV and Bitmap Reconstruction	3.3K (24.2%)
	Pixel Reconstruction	3.9K (28.7%)

한다. 따라서 인코더에 비해 상당히 적은 수의 게이트를 이용한다.

표 3.9는 이전 방식과 제안 방식의 FPGA 구현 결과를 보인다. 합성 기술이 다르긴 하지만 로직 셀(logic cell)을 이용하여 대략적인 하드웨어 크기 비교가 가능하다. 보편적인 BTC에 대한 하드웨어[42] 합성 결과는 표 3.9의 두 번째 행에서 확인할 수 있다. 해당 구조에서는 총 3,112개의 로직 셀이 이용된다. AM-BTC에 대한 하드웨어[28] 합성 결과는 표 3.9의 세 번째 행에서 확인할 수 있다. 해당 구조는 프레임 메모리와 내부 라인 버퍼를 제외하고 16,961 로직 셀을 이용한다. 네 번째, 다섯 번째 행은 Spartan-3 XC3S1500, Virtex-6 XC6VLX75T에 대한 제안 방식의 합성 결과를 보인다. 각 FPGA에서 총 18,461 로직 셀과 17,581 로직 셀이 이용되었다. 제안 방식의 경우 보편적인 BTC 방식에 비해 대략 여섯 배의 로직 셀을 이용한다. 하지만 제안 방식의 처리 속도가 보편적인 BTC 방식에 비해

표 3.9 이전 방식과 제안 방식의 FPGA 합성 결과 비교

Design	Technology	Slice (max)	Logic cell (max)	Memory (bit)
Conv. BTC [42]	Virtex-E XCV200E	1,383 (2,352)	3,112 (5,292)	N/A
AM-BTC [28]	Spartan-3 XC3S2000	7,538 (20,480)	16,961 (46,080)	N/A
Proposed	Spartan-3 XC3S1500	8,205 (13,312)	18,461 (29,952)	2 BRAM (2x18K)
	Virtex-6 XC6VLX75T	2,747 (11,640)	17,581 (74,496)	1 BRAM (1x36K)

훨씬 빠르다. Virtex-6 XC6VLX75T의 경우 제안 인코더 방식의 최대 주파수가 105 MHz로 확인되었다. 따라서 제안 인코더의 처리 속도는 20.2 Gbps이다. 이 수치는 보편적인 BTC 방식의 0.19 Gbps에 비해 훨씬 높은 수치이다. 동일한 동작 주파수가 적용된다고 해도 제안 방식은 보편적인 BTC 방식보다 40배 빠른 처리 속도를 보인다. 이와 같은 특성을 고려할 경우 제안 방식의 로직 셀 수는 그리 큰 편이 아님을 알 수 있다.

VQ-BTC 방식들은 픽셀 클러스터링을 위하여 K-means 클러스터링 또는 PCA 등을 이용한다[26-27]. 그러나 이와 같은 클러스트링 과정은 실시간 처리가 어려운 문제가 있고, 하드웨어 구현에 대한 연구는 아직 확인되지 않았다. AM-BTC 방식은 97 MHz의 주파수에서 동작한다고 설명되어 있으나 매 사이클당 처리되는 픽셀 수가 알려지지 않았다[28]. 따라서 처리 속도와 하드웨어 크기를 구체적으로 비교할 수 없다.

제 4 장 저비용 근접-무손실 프레임 메모리 압축을 위한 고속 1D SPIHT[3]

4.1 인코더 하드웨어 구조

이 장은 고속으로 동작할 수 있는 SPIHT 인코더 하드웨어 구조를 제안한다. 기존 SPIHT 알고리즘은 효율적으로 병렬성을 이용할 수 있도록 수정된다. 이를 위해 SPIHT에 존재하는 의존 관계가 분석되고, 병렬 파이프라인이 가능한 구조가 최종적으로 제안된다.

4.1.1 의존 관계 분석 및 제안하는 파이프라인 스케쥴

제안하는 알고리즘은 웨이블릿(wavelet) 밴드 레벨마다 RP, SP, FRP 등 총 3개의 패스를 수행한다. 그림 4.1 (a)에서 확인할 수 있듯이 기존의 방식에는 병렬 처리를 방해하는 두 가지의 의존 관계가 존재한다. 첫 번째는 SP와 FRP 사이에 존재하는 의존 관계이다. 이 의존 관계는 FRP를 SP에 비해 한 사이클 미뤄 해결할 수 있다. 두 번째는 높은 밴드 레벨의 SP와 낮은 밴드 레벨의 SP 사이에 존재하는 의존 관계이다. 이 의존 관계를 해결하기 위하여 낮은 밴드 레벨의 SP일수록 처리 순서를 한 사이클씩 미루도록 한다.

그림 4.1 (b)는 제안하는 처리 순서를 보인다. 가로축은 DWT 분해 레

[3] This chapter is part of an article, "A High-Throughput Hardware Design of a One-Dimensional SPIHT Algorithm," will be published in the IEEE Transactions on Multimedia (accepted, December 17, 2015, DOI: 10-1109/TMM.2015.2514196).

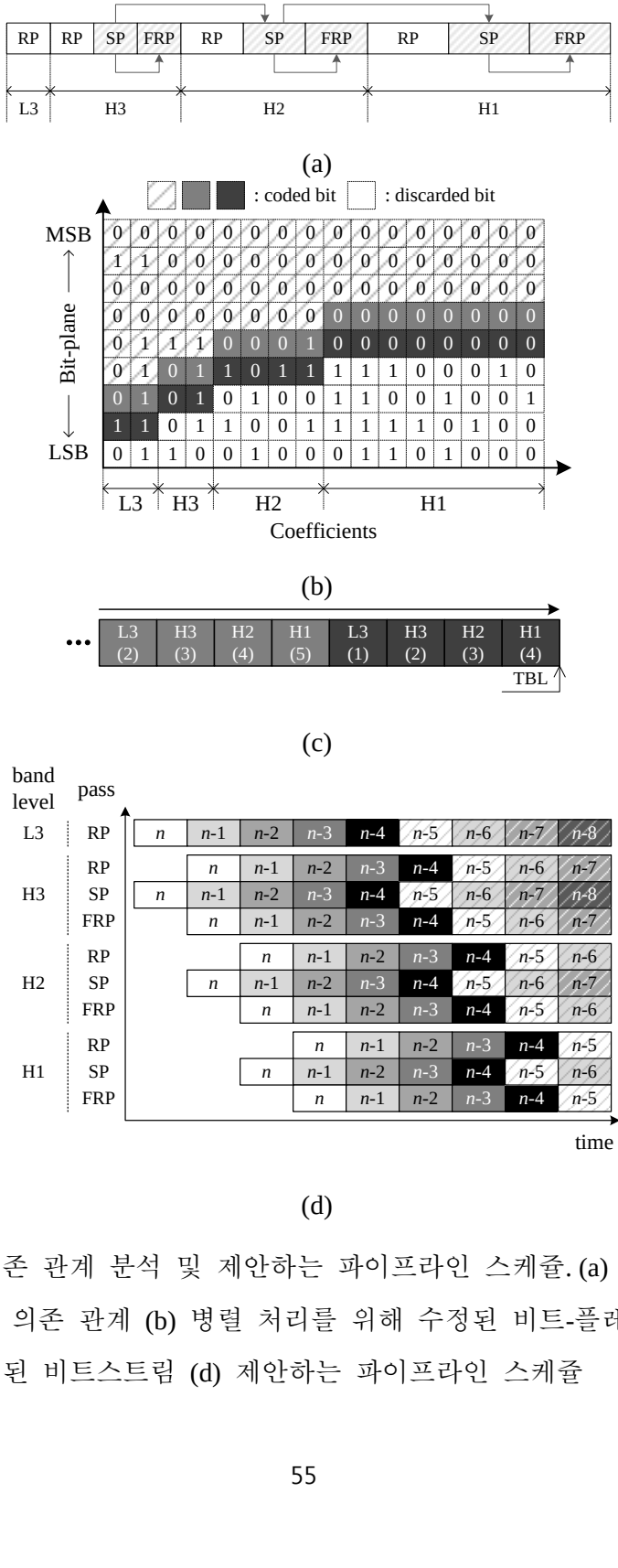


그림 4.1 의존 관계 분석 및 제안하는 파이프라인 스케쥴. (a) 패스 사이에 존재하는 의존 관계 (b) 병렬 처리를 위해 수정된 비트-플레인 예 (c) (b)에서 생성된 비트스트림 (d) 제안하는 파이프라인 스케쥴

벨이 3일 때의 계수들을 보인다. 그리고 세로축은 0번째 비트-플레인에서부터 8번째 비트-플레인까지를 보인다. 어둡게 표시된 영역은 동일한 시간에 처리되는 비트-플레인을 의미한다. 예를 들어 H3의 세 번째 비트-플레인은 H2의 네 번째 비트-플레인, H1의 다섯 번째 비트-플레인과 동시에 코딩된다. 다음 사이클에는 바로 아래인 H3의 두 번째 비트-플레인, H2의 세 번째 비트-플레인, H1의 네 번째 비트-플레인이 병렬적으로 처리된다. 제안 처리 순서에서는 높은 밴드 레벨이 낮은 밴드 레벨에 비해 먼저 코딩을 시작한다. 그 결과, 서로 다른 밴드 레벨의 SP간의 의존 관계가 해결되고, SP와 FRP간의 의존 관계만 남게 된다. 앞서 설명한 바와 같이 SP와 FRP간의 의존 관계는 FRP를 한 사이클 미뤄 해결하도록 한다.

그림 4.1 (b)에서 보인 처리 순서 변화는 비트스트림 포맷(format)에 영향을 미친다. 왜냐하면 비트스트림은 생성되는 순서로 저장되기 때문이다. 그림 4.1 (c)는 L3의 두 번째 비트-플레인, H3의 세 번째 비트-플레인, H2의 네 번째 비트-플레인, H1의 다섯 번째 비트-플레인이 순서대로 저장되었을 때의 비트스트림 모습을 보인다. 그 다음 사이클에는 L3의 첫 번째 비트-플레인, H3의 두 번째 비트-플레인, H2의 세 번째 비트-플레인, H1의 네 번째 비트 플레인이 순서대로 저장된다. H1의 네 번째 비트 플레인을 코딩한 후 생성된 비트스트림의 길이가 목표 비트스트림 길이에 닿았을 경우 H3는 두 번째 비트-플레인의 비트까지 저장하고, H2와 H1은 각각 세 번째, 네 번째 비트-플레인의 비트까지 저장한다. 그림 4.1 (b)의 흰색 영역은 비트스트림에 포함되지 못한 버려진 데이터를 의미한다.

그림 4.1 (b)의 처리 순서는 그림 4.1 (d)의 파이프라인 스케쥴로 확장된다. 첫 번째 사이클(cycle 0)에는 L3의 RP와 H3의 SP가 n번째 비트-플레

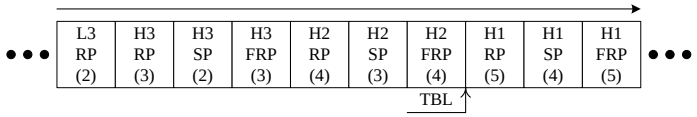
인을 처리한다. 두 번째 사이클에는 H3의 RP, FRP가 n번째 비트-플레인을 처리한다. 또한 H2의 SP가 n번째 비트-플레인을 처리한다. 같은 시간 동안 L3의 RP와 H3의 SP는 n-1번째 비트-플레인을 처리한다. 세 번째 사이클부터는 모든 밴드 레벨의 모든 패스들이 병렬적으로 동작한다. 그 결과 매 사이클마다 단일 비트-플레인에 해당하는 코딩 결과가 생성된다.

제안하는 파이프라인 스케쥴은 의존 관계를 해결하기 위해 제안되었지만 압축 효율 향상에도 도움이 된다. 일반적으로 높은 밴드 레벨은 낮은 밴드 레벨보다 더 중요한 정보를 갖는다. 제안하는 파이프라인 스케쥴의 경우 높은 밴드 레벨에서 먼저 코딩이 시작된다. 그 결과 높은 밴드 레벨이 더 많은 비트를 할당받을 확률이 높아지고, 압축 효율 역시 향상된다. 다음 장에서는 인코더의 압축 효율을 추가적으로 향상시킬 수 있는 방법을 제안하도록 한다.

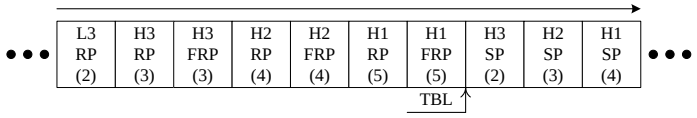
4.1.2 분류 비트 재배치

이전 장에서 제안된 파이프라인 스케쥴은 알고리즘의 빠른 연산을 가능케 하였다. 이 장에서는 인코더의 압축 속도를 늦추지 않으면서 압축 효율을 향상시킬 수 있는 방법을 제안하도록 한다.

제안하는 방식은 RP, FRP, SP에서 출력된 비트스트림의 배치 순서를 바꿔 압축 효율을 향상시킨다. 비트스트림이 생성된 순서대로 저장될 경우 그림 4.2 (a)와 같게 된다. 각각의 사각형 내부 괄호 안의 숫자는 비트-플레인 번호를 의미한다. 예를 들어, 가장 좌측에 있는 ‘L3 RP (2)’는 2번째 비트-플레인에 대한 L3 밴드의 RP를 의미한다. 4 번째 비트-플레인에 대한 H2 밴드의 FRP를 끝으로 목표 비트스트림 길이를 만났다고 가정하



(a)



(b)

그림 4.2 분류 비트 재배치. (a) 생성되는 순서 (b) 재배치된 순서

자. 이 경우 5 번째 비트-플레인에 대한 H1 밴드의 RP, FRP 비트스트림과 4 번째 비트-플레인에 대한 H1 밴드의 SP 비트스트림이 최종 비트스트림에 포함되지 않는다. SP에서 생성된 분류(sorting) 비트의 경우 이진 트리에 대한 중요성 검사 결과 정보를 내포할 뿐 실제로 복원되는 픽셀값 정보는 전혀 가지고 있지 않다. 즉, 해당하는 RP, FRP의 비트가 생성되어야 의미를 갖는다. 이 예에서 3 번째 비트-플레인에 대한 SP의 분류 비트는 3 번째 비트-플레인에 대한 RP, FRP가 진행될 때에만 의미를 갖는다. 그러나 RP, FRP에 해당하는 비트가 목표 비트스트림 길이 제한에 걸려 코딩되지 않으므로 3 번째 비트-플레인에 대한 SP의 분류 비트는 존재 의미가 없다. 이와 같은 문제를 해결하기 위해 본 논문은 SP에서 출력된 분류 비트를 RP, FRP에서 출력된 크기 비트 다음으로 위치시킨다. 단, 이 작업은 매 사이클 단위로 진행된다. 분류 비트가 크기 비트 다음에 위치하므로 최종 비트스트림에 분류 비트 대신 크기 비트가 포함될 확률이 증가한다. 제안된 순서는 그림 4.2 (b)에서 확인할 수 있다. 3 번째 비트-플레인에 대한 H2의 SP 비트는 5 번째 비트-플레인에 대한 H1의 RP, FRP

비트 다음으로 옮겨진다. 해당 예에서 5 번째 비트-플레인에 대한 H1의 FRP 비트를 끝으로 코딩이 중단된다. 그림 4.2 (a)와는 달리 5 번째 비트-플레인에 대한 RP, FRP 비트가 최종 비트스트림에 포함되고, 두 번째 비트-플레인에 대한 H3의 SP, 세 번째 비트-플레인에 대한 H2의 SP 비트가 최종 비트스트림에서 제외된다. 새롭게 포함된 RP, FRP 비트는 화질에 긍정적인 효과만을 주고, 제외된 SP 비트는 부정적 효과를 주지 않는다.

4.2 디코더 하드웨어 구조

이 장은 디코더 하드웨어 구조를 위한 SPIHT 알고리즘을 제안한다. 디코더 하드웨어 구조에서는 의존 관계 문제뿐만 아니라 각 패스가 접근하는 비트스트림의 시작 주소를 계산하는 문제도 해결되어야 한다. 이어지는 4.2.1 장에서는 비트스트림의 시작 주소를 계산하는 방법을 제안하고, 4.2.2 장에서는 디코더 측면에서 압축 효율을 향상시킬 수 있는 방법을 제안하도록 한다.

4.2.1 비트스트림의 시작 주소 계산

디코더 구조 설계에서 해결해야 하는 중요한 문제 중 하나는 각 패스의 시작 주소를 미리 알 수 있어야 한다는 점이다. 이는 병렬로 동작하는 패스들이 각각의 비트스트림에 동시 접근해야 하기 때문이다.

이 문제를 다루기에 앞서 각 패스의 시작 주소를 미리 아는 것이 어려운 이유를 분석하도록 한다. 이를 위해 각 패스에서 생성되는 비트스트림의 길이를 먼저 확인하도록 한다. 표 4.1은 SP, FRP, RP에서 생성되는

표 4.1 상태 변화에 따른 비트 수 변화

Pass	Present state	Significance test results		Next state	Generated bit
SP	LISa	-		LISa or LISb	1 (sorting)
	LISb	1	-	LISa	3 (sorting)
		0	-	LISb	1 (sorting)
FRP	LISa	0 / 0		LIP / LIP	2 (magn.)
		0 / 1		LIP / LSP	2 (magn.) + 1 (sign)
		1 / 0		LSP / LIP	2 (magn.) + 1 (sign)
		1 / 1		LSP / LSP	2 (magn.) + 2 (sign)
RP	LIP	0		LIP	1 (magn.)
		1		LSP	1 (magn.) + 1 (sign)
	LSP	-		LSP	1 (magn.)

비트스트림의 모든 경우를 보인다. 표 4.1의 첫 번째 열은 패스의 종류를 보인다. 두 번째 열과 세 번째 열은 각각 현재 상태와 상태 변화를 야기하는 중요성 검사 결과를 보인다. 네 번째 열은 중요성 검사 이후의 다음 상태를 보인다. 마지막으로 다섯 번째 열은 생성된 비트를 보인다.

SP의 경우 현재 상태는 LISa 또는 LISb이다. 현재 상태가 LISa인 경우 중요성 검사 결과와 상관없이 하나의 분류 비트가 출력된다. 반면에 현재 상태가 LISb인 경우 두 가지의 중요성 검사(그림 2.4의 라인 11과 라인 25)가 존재하고, 그림 2.4의 라인 25에서 보인 중요성 검사 결과에 따라 생성되는 비트 수가 달라진다. 라인 25에서의 중요성 검사가 '1'을 출력하면 상태는 LISb에서 LISa로 변화한다. 그리고 2 비트 크기의 추가 분류 비트가 생성된다(그림 2.4의 라인 10). 그 결과 총 3 비트의 분류 비트가 생성된다. 반면에 라인 25에서의 중요성 검사가 '0'을 출력하면 추가 비트 생성없이 다음 상태는 계속 LISb에 머물게 된다(그림 2.4의 라인

24). 이와 같이 SP에서 생성되는 비트 수는 중요성 검사 결과에 따라 1 또는 3으로 달라진다. 따라서 SP에서 생성되는 비트스트림의 길이는 직접 디코딩하지 않고는 미리 예측할 수가 없다. 이 문제를 해결하기 위해서는 SP에서 생성된 비트스트림의 길이를 미리 계산할 수 있는 방법이 필요하다.

FRP의 현재 상태는 항상 LISa 상태이고, 집합 내부의 두 계수가 픽셀 단위의 검사를 받는다. 이때, 두 계수의 크기에 따라 (LIP, LIP), (LIP, LSP), (LSP, LIP), (LSP, LSP) 등 총 4 가지의 상태 변화가 가능하다. 크기 비트는 모든 경우에서 2 비트씩 생성된다. 반면에 부호 비트는 상태가 LSP로 바뀔 때에만 생성된다. 그러므로 비트스트림의 길이는 미리 계산될 수 없다. 이 문제를 해결하기 위해서는 FRP에서 코딩된 비트스트림의 길이를 미리 계산할 수 있는 방법이 필요하다.

RP의 경우 현재 상태는 LIP 또는 LSP이다. 현재 상태가 LIP이고 중요성 검사 결과가 ‘0’인 경우 다음 상태도 LIP로 유지된다. 이 경우 하나의 크기 비트만 출력된다. 반면에 LIP가 LSP로 변하는 경우 하나의 크기 비트와 하나의 부호 비트 등 총 2 비트를 출력한다. 현재 상태가 LSP인 경우 다음 상태는 항상 LSP가 되고, 하나의 크기 비트가 항상 생성된다. 그러므로 생성된 비트스트림의 길이는 현재 상태와 중요성 검사 결과에 따라 1 또는 2가 될 수 있다. 즉, RP에서 코딩되는 비트스트림의 길이는 미리 예측될 수 없다. 이 문제는 FRP에서의 문제와 동일하므로 FRP와 동일한 방법으로 문제를 해결하고자 한다.

앞서 보인 분석에서 비트스트림의 길이가 중요성 검사 결과에 따라 변함을 보였다. 이러한 의존 관계는 각 패스에서 디코딩하게 될 비트스트

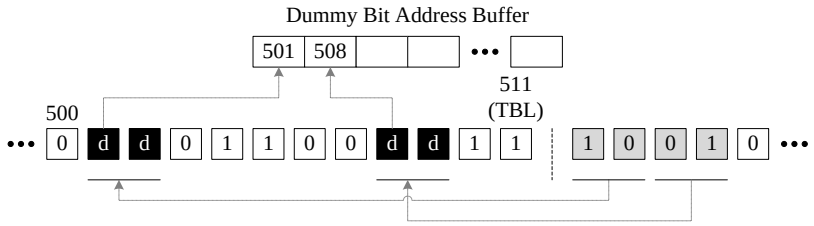


그림 4.3 더미 비트 생성 및 재사용

림의 길이를 실제 디코딩 과정 없이 미리 구하기가 불가능하다는 사실을 의미한다. 이어지는 내용에서는 SP와 RP(또는 FRP)에서 디코딩하게 될 비트스트림의 길이를 미리 계산할 수 있는 방법을 제안하도록 한다.

SP에서 생성된 비트스트림의 길이를 미리 계산하기 위하여 현재 상태가 LISb인 경우 항상 3 비트가 출력되도록 SP의 동작을 수정하도록 한다. 다시 말해, 현재 상태가 LISb이고 중요성 검사가 '0'을 출력할 때 두 개의 더미 비트가 하나의 분류 비트와 더불어 출력된다. SP에서 출력되는 두 개의 더미 비트는 정보를 포함하지 않기 때문에 압축 효율을 감소시킨다. 이와 같은 더미 비트 낭비를 피하기 위하여, 추후 생성된 비트를 더미 비트 자리에 저장한다.

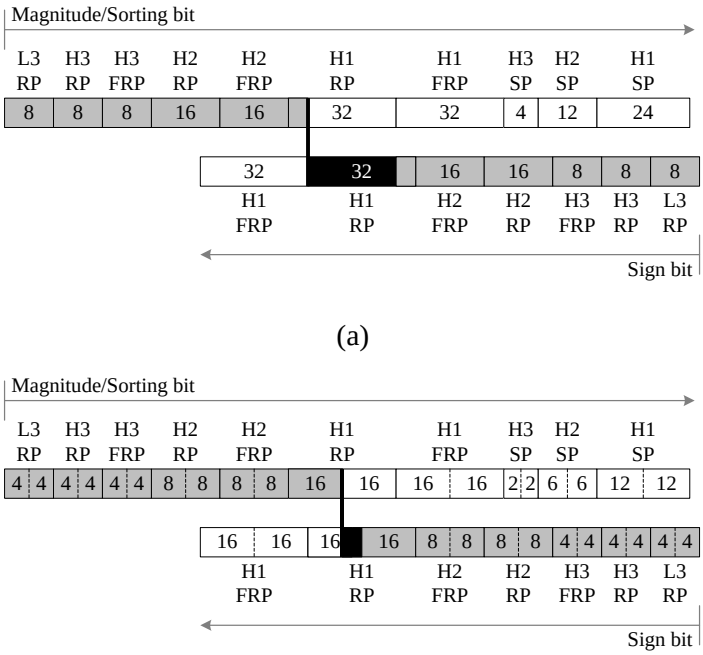
제안한 더미 비트 재사용 방식은 그림 4.3에서 확인 가능하다. 우선 더미 비트의 위치를 저장하기 위한 버퍼가 추가된다. 이 예에서는 더미 비트가 주소 501과 주소 508에 저장이 된다. 이때, 각각의 위치는 2개의 더미 비트를 저장한다. 더미 비트 주소 버퍼에 저장된 위치 정보는 추후 생성된 비트를 저장하는데 이용된다. 이 예에서는 목표 비트스트림 길이가 512이다. 따라서 512 번째 비트와 513 번째 비트는 원래 버려진다. 그러나 더미 비트 재사용 방식을 이용할 경우 해당 비트들은 각각 501 번째, 502 번째 위치에 저장이 된다. 마찬가지로 514 번째, 515 번째 비트는

508 번째, 509 번째 위치에 저장이 된다. 이와 같은 방식을 통해 버려지는 비트가 사실상 사라지게 된다. 따라서 SP의 더미 비트에 의한 화질 열화 역시 방지된다. 디코더는 SP에서 LISb가 유지될 경우 더미 비트가 발생한다고 판단한다. 따라서 더미 비트 위치에 있는 비트는 디코더의 버퍼 공간에 따로 저장이 되고, 추후 디코딩된다. 제안하는 더미 비트 재사용 방식은 추가 메모리와 회로를 요구한다. 메모리 및 회로의 크기는 4.3장에서 설명하도록 한다.

RP와 FRP의 경우 생성된 비트스트림의 길이를 미리 계산하기 위한 방법이 BPS[33]에 소개되었다. RP와 FRP의 경우 출력되는 크기 비트의 수가 중요성 검사 결과와 상관없이 항상 1과 2로 고정되어 있다. BPS는 크기 비트와 부호 비트가 저장되는 위치와 방향을 반대로 설정하였다. 다시 말해, 크기 비트는 좌측에서 우측 방향으로 저장되고, 부호 비트는 우측에서 좌측 방향으로 저장된다. 따라서 부호 비트와 무관하게 크기 비트의 길이를 미리 계산할 수 있다. 부호 비트의 경우 크기 비트에 따라 그 길이가 달라진다. 따라서 부호 비트에 대한 디코딩을 크기 비트에 대한 디코딩보다 한 사이클 늦추도록 한다.

4.2.2 절반-패스 처리 방법

이전 장에서 언급한 바와 같이 본 논문에서는 디코더의 병렬 처리를 위해 크기 비트와 부호 비트를 따로 저장한다. 하지만 분리된 크기 비트와 부호 비트는 압축 효율을 감소시키는 새로운 문제를 발생시킨다. 이 문제는 한 패스에 포함된 계수의 수가 많을수록 더 심한 화질 열화 현상을 발생시킨다. 그림 4.4는 압축 효율이 감소하는 한 예를 보인다. 이 예



(a)

(b)

그림 4.4 절반 패스 처리 방법의 예. (a) 기존 패스 (b) 절반 패스

에서 DWT 분해 레벨은 3으로 설정되었고, 코딩 블록 크기는 1×64이다. 각각의 사각형은 각각의 패스를 의미하고, 사각형 내부의 숫자는 해당 패스에서 생성될 수 있는 최대 비트 수를 의미한다. 앞서 설명한 바와 같이 크기 비트는 좌측에서 우측으로 저장이 되고, 부호 비트는 반대 반향으로 저장이 된다. 따라서 생성된 비트스트림의 길이가 목표 비트스트림의 길이에 가까워지면 크기 비트와 부호 비트가 결국 겹치게 된다. 그림 4.4 (a)에서 확인할 수 있듯이 H1의 RP에서 생성된 크기 비트는 대응되는 부호 비트와 겹친다. 일반적으로 부호 비트가 크기 비트에 비해 더 중요하므로 본 논문에서는 부호 비트만 최종 비트스트림에 포함시키고, 크기 비트는 제외시키도록 한다. 하지만 부호 비트는 대응되는 크기 비트가 비트스트

림에 존재하지 않으면 아무 정보도 제공하지 못한다. H1의 RP에서 크기 비트와 부호 비트가 겹칠 경우 버려지는 비트 수는 최대 32이다. 이와 같이 버려지는 비트는 화질 열화에 큰 영향을 미칠 수 있다.

버려지는 비트 수를 감소시키기 위하여 본 논문에서는 패스의 크기를 반으로 나눠 병렬로 처리하도록 한다. 그림 4.4 (b)는 제안하는 절반-패스 처리 방법을 보인다. 이 예에서 H1에 대한 RP 동작은 두 개의 RP 동작으로 나눠지고, 나눠진 각각의 RP는 H1에 포함된 계수들을 절반씩(16개씩) 처리한다. 이 경우 나눠진 패스에서 생성될 수 있는 최대 비트 수는 16이다. 따라서 크기 비트와 부호 비트가 겹침으로써 버려질 수 있는 비트 수가 절반이 된다. 이와 같은 병렬 처리 방식은 각 패스의 시작 주소를 계산하기 위한 하드웨어 비용을 추가적으로 요구한다. 따라서 본 논문에서는 절반보다 더 작은 단위로 패스를 나누지 않도록 한다.

4.3 하드웨어 구현

이 장은 제안한 인코더, 디코더에 대한 하드웨어 구현 방식 및 결과를 보인다. 제안한 알고리즘을 하드웨어로 구현하기 위하여 Verilog HDL을 이용하였다. 제안 하드웨어의 경우 내부 버퍼 크기를 목표 압축률 2에 대한 크기로 설정하였기 때문에 2 이상의 압축률이 지원가능하다. 그리고 이용하는 블록 크기는 1×64이며, DWT 분해 레벨은 3이다. 제안 하드웨어는 다른 블록 크기, 분해 레벨에 대하여 쉽게 수정이 가능하다. 참고로 이 장에서는 다른 하드웨어와의 공정한 비교를 위해 흑백 이미지에 대한 구현 결과를 보이도록 한다.

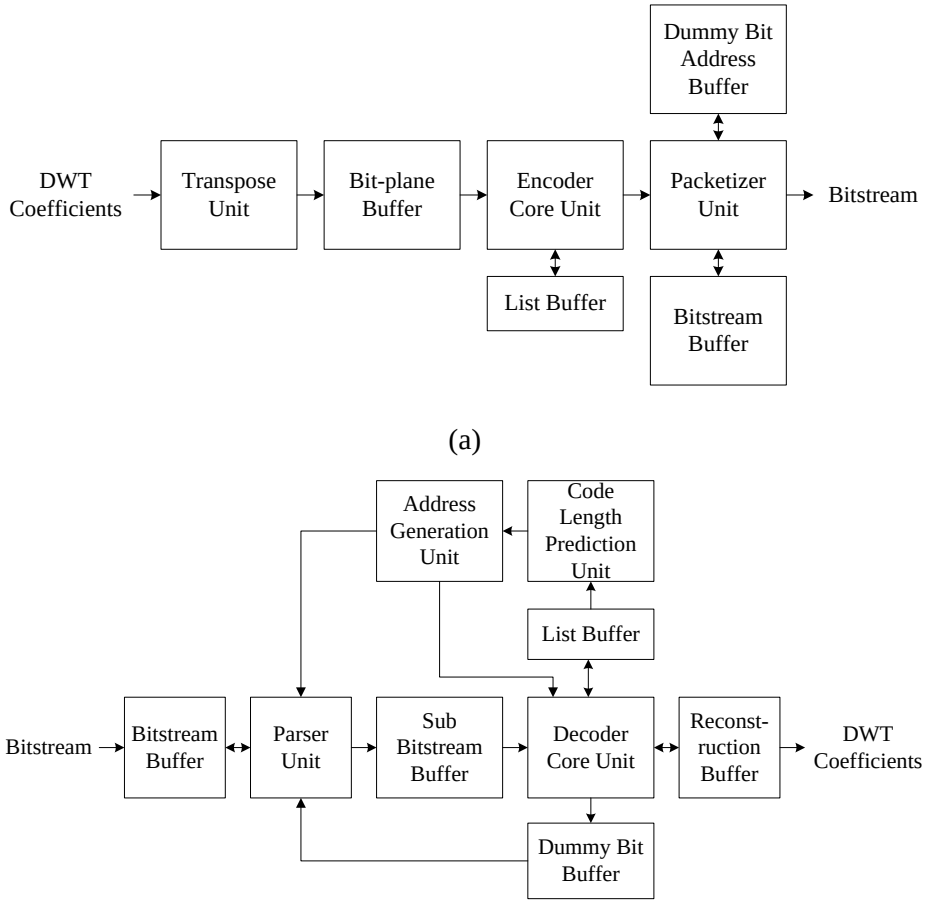


그림 4.5 제안하는 1D SPIHT 하드웨어의 블록 다이어그램.
(a) 인코더 (b) 디코더

그림 4.5 (a)는 제안하는 1D SPIHT 인코더 하드웨어 구조를 보인다. 인코더의 입력 데이터는 DWT 계수이다. 변환 유닛은 이 데이터를 계수 단위에서 비트-플레인으로 바꿔 인코더 코어 유닛으로 전달한다. 제안 구조는 매 사이클마다 하나의 비트-플레인을 처리한다. 이를 위해 인코더

코어 유닛은 RP 8개, FRP 6개, SP 6개 등 총 20개의 처리 유닛을 포함한다. 더미 비트 주소 버퍼는 인코더 코어 유닛과 패킷타이저 유닛에서 생성된 더미 비트의 주소를 저장한다. 본 논문에서는 더미 비트 주소 버퍼의 크기를 160 비트로 설정하였다. 리스트(list) 버퍼는 각 계수마다 하나씩 갖게 되는 Null, LIP, LSP, LISa, LISb 등의 인코딩 상태를 저장한다. 패킷타이저 유닛은 여러 패스에서 생성된 비트스트림을 미리 결정한 포맷에 맞게 묶는다. 그리고 해당 모듈은 발생한 더미 비트의 수를 세고, 추후 이 비트 공간을 이용한다.

제안하는 1D SPIHT 디코더 하드웨어 구조는 그림 4.5 (b)와 같다. 디코딩할 입력 비트스트림은 비트스트림 버퍼에 저장된다. 파서 유닛은 각각의 패스에서 처리될 비트스트림을 분리하고, 이들을 하위(sub)-비트스트림 버퍼에 저장한다. 코드 길이 예측 유닛과 주소 생성 유닛은 각 패스가 디코딩을 시작할 주소를 미리 계산하고, 이 정보를 파서 유닛에 전달한다. 인코더 코어 유닛과 마찬가지로 디코더 코어 유닛은 20개의 처리 유닛을 포함한다. 그러나 RP와 FRP에서 부호 비트 디코딩이 한 사이클 미뤄지기 때문에 14개의 처리 유닛이 추가되어 총 34개의 유닛이 병렬적으로 동작한다. 디코더의 리스트 버퍼 크기는 384 비트로 인코더의 리스트 버퍼 크기의 두 배이다. 이는 부호 비트 디코딩이 한 사이클 미뤄지기 때문에 그러하다. 더미 비트 위치에 포함된 비트는 임시적으로 더미 비트 버퍼에 저장이 되고, 추후 목표 비트스트림 길이 이후의 디코딩에 이용된다. 흑백 이미지 압축의 경우 재사용되는 더미 비트의 횟수를 20으로 설정하였으며, 이는 4:2:2 컬러 이미지 압축에 비해 절반의 크기이다. 그러므로 더미 비트 버퍼의 크기는 40(=20×2) 비트이다. 디코딩이 완료된 계수는 복

표 4.2 이전 방식과 제안 방식의 FPGA 합성 결과 비교

Architecture	*No List SPIHT [30] **	*Bit-plane Parallel SPIHT [32] **	*BPS [33]*		*Proposed*	
			Encoder	Decoder	Encoder	Decoder
Algorithm	2D SPIHT	2D SPIHT	2D SPIHT		1D SPIHT	
Technology	Xilinx Virtex 2-4	Xilinx Virtex 2000E	Xilinx Virtex 5-LX330		Xilinx Virtex 6-LX75T	
Frequency (MHz)	100	56	150	110	115	139
Throughput (Gbits/sec)	0.29	3.58	2.34	1.72	5.66	6.35
Normalized Throughput (Mbits/(sec x MHz)	2.9	63.9	15.6	15.6	49.2	45.7
Slice (no.)	4,500	11,904+6,528 +18,816	3,592	3,421	2,597	4,035
Logic Cell (no.)	10,125 / 11,520	83,808 / 43,200	22,996 / 331,876	21,901 / 331,876	16,621 / 74,496	25,824 / 74,496
Logic Cell / Normalized Throughput (no. x (sec x MHz) / Mbits)	3,491	1,312	1,474	1,404	338	565
Internal Memory (Kbit)	432 (24 BRAM)	N/A	8.3	6.7	0	0

* Encoder only

원 버퍼에 저장되고, 외부 하드웨어에 의해 접근 가능하다.

이 문단부터는 이전 방식과 제안 방식의 성능 및 하드웨어 비용을 비교하도록 한다. 비교를 위하여 제안 하드웨어는 Xilinx Virtex 6-LX75T을 목표로 합성되었다. 표 4.2는 이전 방식과 제안 방식의 FPGA 구현 결과를 보인다. 제안 인코더와 디코더 결과는 각각 표 4.2의 여섯 번째 열과 일곱 번째 열에서 확인할 수 있다. 표 4.2의 다섯 번째 행에서 보인 바와 같이 제안 인코더와 디코더 구조의 최대 동작 주파수는 115MHz와

139MHz이다. 이 주파수에서 제안 인코더와 디코더의 처리 속도는 각각 5.66Gbps와 6.35Gbps이다. 합성 기술에 무관한 비교를 위해 표 4.2의 일곱 번째 행은 이전 방식과 제안 방식의 정규화된 처리 속도를 보인다. [32]의 인코더 구조는 가장 높은 정규화된 처리 속도를 보인다. 그러나 하드웨어 크기가 너무 큰 단점이 있다. [33]은 인코더, 디코더 모두에서 상당히 우수한 정규화된 처리 속도를 보인다. 그러나 제안 방식은 [33]에 비해 3.15배 높은 정규화된 처리 속도를 보인다. [33]은 4×4 크기의 단일 비트-플레인(16 비트)을 매 사이클마다 처리한다. 제안 방식의 경우 1×64 크기의 단일 비트-플레인(1×64 비트)을 매 사이클마다 처리한다. 따라서 동작 주파수가 같은 경우 제안 방식의 속도가 [33]에 비해 4배 빠를 것으로 예상할 수 있다. 그러나 실제 속도 향상이 4배가 되지 않는 이유는 그림 4.1 (d)에서 보인 제안 방식의 파이프라인 구조가 10 비트-플레인을 13 사이클 동안 처리하기 때문이다.

[33]과는 달리 제안 방식은 블록 크기를 증가시켜 더 빠른 처리 속도를 보일 수 있다. 이때의 처리 속도 증가는 블록 크기에 비례한다. 이는 제안 방식이 여러 주파수 밴드를 동시에 처리할 수 있기 때문이다. 반면에 [33]은 고정된 4×4 크기를 처리하므로 더 이상 처리 속도를 향상시킬 수 없다. 이는 [33]이 하나의 주파수 밴드 내에서만 동작하고, 여러 주파수 밴드에 존재하는 의존 관계를 해결하지 못했기 때문이다.

이전 방식과 제안 방식의 목표 기술이 다르지만 표 4.2의 아홉 번째 행에서 보인 등가 로직 셀 수를 이용하여 하드웨어의 크기를 비교할 수 있다. 제안 인코더와 디코더 구조는 각각 16,621 로직 셀과 25,824 로직 셀을 이용한다. 표 4.2의 열 번째 행은 정규화된 처리 속도 당 이용하는

표 4.3 제안 하드웨어의 처리 속도 (ASIC)

Module	Frequency (MHz)	Throughputs (Gbits/sec)
Encoder	143	7.04
Decoder	167	7.63

로직 셀 수를 보이는데, 제안하는 구조가 가장 작은 값을 보이는 것을 확인할 수 있다. [33]과 비교했을 때, 제안 인코더, 디코더는 동일한 정규화된 처리 속도를 얻기 위하여 각각 22.93%와 40.24%의 로직 셀만을 이용한다. 제안 구조의 내부 메모리는 모두 레지스터로 합성되었기 때문에 표 4.2의 마지막 행에서 보인 바와 같이 그 크기가 0이다.

이어지는 내용은 ASIC 합성을 목표로 한 제안 하드웨어의 구현 결과를 보인다. 작성한 Verilog 모델은 Synopsys Design Compiler로 합성되었으며, Silterra 0.13 μm 공정 라이브러리가 이용되었다. 표 4.3에서 보인 바와 같이 제안 인코더, 디코더 구조의 최대 동작 주파수는 각각 143 MHz와 167 MHz이다. 그리고 이때의 처리 속도는 각각 7.04 Gbps와 7.63 Gbps이다. 최대 동작 주파수의 경우 0.13 μm보다 더 발전된 공정 라이브러리를 이용할 경우 그 크기가 더 증가될 수 있다. 제안 구조를 65 nm 공정으로 합성할 경우 인코더와 디코더의 최대 동작 주파수는 각각 526 MHz와 588 MHz이다. 참고로 인코더 구조의 임계 경로는 패킷타이저 유닛 내부에 존재하고, 디코더 구조의 임계 경로는 코드 길이 예측 유닛과 파서 유닛 사이에 존재한다.

표 4.4는 0.13 μm 공정 라이브러리를 이용했을 때 제안 하드웨어가 이용하는 게이트 수와 메모리 크기를 보인다. 인코더 구조의 전체 게이트 수는 37.2K이다. 이 중 변환(transpose) 유닛의 게이트 수가 6.4K이고, 이

표 4.4 제안 하드웨어의 게이트 수와 메모리 크기 (ASIC)

Module		Gate count	Memory-bits (corresponding gate count)	Total gate count
Proposed Encoder	*Total*	37.2K	1,984 (12.8K)	50.0K
	Forward DWT Unit	3.3K	-	-
	Transpose Unit	6.4K	-	-
	Encoder Core Unit	13.9K	-	-
	Packetizer Unit	13.6K	-	-
	Bit-plane Buffer	-	1,120 (7.7K)	-
	Bitstream Buffer	-	512 (3.4K)	-
	Dummy Bit Address Buffer	-	160 (0.7K)	-
	List Buffer	-	192 (1.0K)	-
Proposed Decoder	*Total*	54.1K	2,216 (14.7K)	68.8K
	Inverse DWT Unit	5.4K	-	-
	Parser Unit	12.3K	-	-
	Sub-bitstream Buffer	1.0K	-	-
	Code Length Prediction Unit	1.0K	-	-
	Address Generation Unit	2.0K	-	-
	Decoder Core Unit	32.4K	-	-
	Bitstream Buffer	-	512 (3.4K)	-
	List Buffer	-	384 (2.2K)	-
	Dummy Bit Buffer	-	40 (0.2K)	-
	Reconstruction Buffer	-	1,280 (8.9K)	-

는 인코더 전체의 17.2%를 차지한다. 그리고 패킷타이저 유닛이 전체 인코더의 36.6%인 13.6K 게이트를 이용한다. 패킷타이저 유닛은 비트-너비가 큰 시프터를 많이 이용하는데, 그 결과 많은 게이트 수를 보이게 되었다. 디코더 구조의 전체 게이트 수는 54.1K이다. 이 중 파서 유닛이 전체 디코더의 22.7%인 12.3K를 이용한다. 인코더의 패킷타이저 유닛과 마찬가

지로 디코더의 파서 유닛은 다수의 시프터에 의해 많은 게이트 수를 이용하게 되었다. 디코더 코어 유닛의 게이트 수는 32.4K로 전체 디코더의 59.9%를 차지한다. 디코더 코어 유닛은 총 34개의 패스를 병렬적으로 동작시킨다. 따라서 많은 게이트 수가 이용된다. 4.2.2장에서 제안한 절반-패스 처리 방식의 경우 동일한 두 회로를 이용한다. 따라서 하나의 회로가 나눠진 두 패스에 의해 공유될 수 있고, 게이트 수도 대략 반까지 감소될 수 있다. 대신 이 경우 한 비트-플레인을 처리하는데 두 사이클이 걸리게 되어 처리 속도가 느려진다.

표 4.4의 네 번째 열은 제안 하드웨어가 이용하는 메모리 크기를 보인다. 괄호 안의 숫자는 해당 메모리가 레지스터로 합성되었을 때의 게이트 수를 의미한다. 비트-플레인 버퍼는 1,120 비트를 이용한다. 그리고 더미 비트 주소 버퍼와 리스트 버퍼가 각각 160 비트, 192 비트씩을 이용한다. 마지막으로 인코더의 비트스트림 버퍼 크기는 512 비트이다. 따라서 인코더에서 이용되는 메모리 크기는 총 1,984 비트이다. 디코더의 경우 비트스트림 버퍼가 512 비트를 이용하며, 복원 버퍼가 1,280 비트 크기이다. 그리고 더미 비트 버퍼와 리스트 버퍼의 크기가 각각 40 비트와 384 비트이다. 따라서 디코더에서 이용되는 전체 메모리의 크기는 2,216 비트이다. 표 4.4의 마지막 열은 메모리가 레지스터로 합성되었을 때의 전체 게이트 수를 보인다. 인코더의 전체 게이트 수는 50.0K이고, 디코더의 전체 게이트 수는 68.8K이다. 이때, 메모리는 인코더의 게이트 수를 34.4% 만큼, 디코더의 게이트 수를 27.2%만큼 증가시킨다.

4.4 실험 결과

본 논문은 고속 1D SPIHT 하드웨어를 위하여 본래의 SPIHT 알고리즘을 수정하였고, 그 결과 압축 효율이 감소하였다. 따라서 압축 효율을 향상시키기 위한 추가 방법을 4.1.2장, 4.2.2장 등에서 제안하였다. 이 장은 압축 효율을 평가하기 위한 실험 결과를 보이도록 한다. 대부분의 실험에서 목표 압축률, 블록 크기, DWT 분해 레벨 등은 3, 1×64, 3으로 설정되었으나 다른 조건이 이용되는 경우 그 값을 따로 표기하도록 한다. 이 장에서는 화질을 측정하기 위해 24개의 Kodak 컬러 이미지[41]을 이용하였다. Kodak 컬러 이미지의 R, G, B 성분은 Y, Cb, Cr 성분으로 변환된 후 4:2:2 포맷으로 서브-샘플링되어 제안 알고리즘에 전달된다.

표 4.5는 제안하는 고속 하드웨어 구조와 4.1.2장, 4.2.2장 등에서 제안한 압축 효율 향상 방법들의 효과를 보인다. 압축 효율을 평가하기 위해 평균 PSNR 수치가 이용되었다. 표 4.5의 두 번째 행은 [6]의 기존 1D SPIHT에 그림 4.1 (b)의 비트-플레인 처리 순서를 적용하였을 때에 대한 참조 소프트웨어의 성능을 보인다. 이와 같은 수정은 제안하는 하드웨어 구조와의 공정한 비교를 위해 수행되었다. 이용된 목표 압축률, 블록 크

표 4.5 화질에 대한 제안 방식들의 효과

Algorithm	PSNR (dB)	⊿PSNR (dB)
Previous [6]	44.81	-
High-throughput hardware implementation	43.41	-1.40
+ Dummy bit reuse	44.03	+0.62
+ Partitioned pass processing	44.22	+0.19
+ Relocation of sorting bits	44.37	+0.15

기, DWT 분해 레벨은 제안 구조에서 이용한 값과 동일하다. 표 4.5의 세 번째 행은 압축 효율 향상 방법을 적용하지 않고, 4.1.1장과 4.2.1장 등에서 제안한 하드웨어 구조를 적용하였을 때의 결과를 보인다. 이 경우 고속 하드웨어 동작이 가능하지만 평균 PSNR은 1.40 dB만큼 감소한다. 네 번째 행부터 여섯 번째 행은 4.1.2장, 4.2.1장, 4.2.2장 등에서 제안한 압축 효율 향상 방법들의 효과를 보인다. 네 번째 행은 그림 4.3에서 보인 더미 비트 재사용 방법에 의해 평균 PSNR이 0.62 dB 향상됨을 보인다. 4.2.2장에서 제안한 절반-패스 처리 방식은 다섯 번째 행에서 보인 바와 같이 평균 압축률을 0.19 dB만큼 향상시킨다. 4.1.2장에서 제안한 분류 비트 재배치 방식은 여섯 번째 행에서 보인 바와 같이 평균 PSNR을 0.15dB만큼 향상시킨다. 제안한 세 가지 압축 효율 방식을 적용할 경우 평균 PSNR은 총 0.96 dB만큼 향상된다. 표 4.5의 두 번째 행(소프트웨어 결과)과 비교했을 때 여섯 번째 행(최종 하드웨어 결과)은 0.44 dB만큼 감소된 평균 PSNR 결과를 보인다.

그림 4.6은 제안한 방식에 의해 복원된 이미지를 보인다. 실험에는 상대적으로 많은 가장자리 정보를 포함하여 압축이 까다로운 Kodak14 이미지의 일부분이 이용되었다. 그림 4.6 (a)는 원본 이미지를 보이고, 그림 4.6 (b)와 그림 4.6 (c)는 [6]과 제안 방식에 의해 복원된 이미지를 보인다. 그림 4.6에서 확인할 수 있듯이 압축이 어려운 이미지였음에도 원본 이미지와 복원된 이미지의 차이를 구분하기가 상당히 어려움을 알 수 있다.

다음은 이전 알고리즘과의 성능 비교에 대한 결과를 보이도록 한다. 표 4.6은 여섯 개의 실험 이미지에 대한 압축 결과와 24개 Kodak 이미지에 대한 평균 압축 결과를 보인다. 처음의 두 결과는 2D SPIHT 알고리즘

(a)

(b)

(c)

그림 4.6 Kodak14 (508, 381) ~ (603, 444) 이미지. (a) 원본 이미지 (b) [6]의 복원 이미지 (c) 제안 방식의 복원 이미지

표 4.6 2D SPIHT 알고리즘과의 압축 효율 비교

Test image	2D SPIHT [4]				2D BPS [33]				Proposed 1D SPIHT	
	Y:Cb:Cr 4:4:4		Y:Cb:Cr 4:2:0		Y:Cb:Cr 4:4:4		Y:Cb:Cr 4:2:0		Y:Cb:Cr 4:2:2	
	PSNR (dB)	CR	PSNR (dB)	CR	PSNR (dB)	CR	PSNR (dB)	CR	PSNR (dB)	CR
Kodak03	48.36	3.05	46.76	4.09	48.24	3.05	46.75	4.07	47.74	3.07
Kodak08	42.99	3.00	44.02	3.11	42.94	3.00	43.97	3.11	40.02	3.01
Kodak09	46.36	3.00	46.88	3.75	46.25	3.00	46.87	3.73	45.67	3.02
Kodak12	48.03	3.00	49.14	3.77	47.91	3.00	49.12	3.75	47.13	3.02
Kodak13	41.75	3.00	42.80	3.04	41.71	3.00	42.75	3.04	40.88	3.01
Kodak14	44.59	3.00	43.14	3.28	44.50	3.00	43.13	3.27	43.40	3.02
Average	45.39	3.02	45.68	3.57	45.31	3.02	45.65	3.55	44.37	3.03

인 [4]와 [33]의 결과이다. [4]의 경우 소프트웨어로 구현되었고, [33]의 경우 하드웨어로 구현되었다. 이때, 이용된 블록 크기는 16×16이고, DWT 분해 레벨은 3으로 설정되었다. 컬러 코딩 방식의 경우 세 컬러 성분의 비트스트림이 비트-플레인 단위로 함께 생성 및 저장된다. 마지막 결과는 고속 하드웨어 구조를 위해 본 논문에서 제안한 1D SPIHT 알고리즘에 의한 결과이다. 2D SPIHT 알고리즘의 경우 입력 이미지 포맷이 4:4:4 또는 4:2:0으로 선택되었다. 1D SPIHT과 같이 4:2:2 포맷이 선택되지 못한 이유는 코딩 블록이 정사각형 형태가 되지 못하기 때문이다. 표 4.6을 통해 1D SPIHT과 2D SPIHT의 컬러 포맷이 다름에도 2D SPIHT이 모든 이미지에 대해 더 좋은 압축 효율을 보이는 것을 알 수 있다. 이는 예상 가능한 결과이다. 왜냐하면 2D SPIHT의 경우 1D SPIHT과는 달리 가로 방향뿐만 아니라 세로 방향에 대해서도 공간적 연관성을 이용할 수 있기 때문이다. 그러나 크기가 큰 라인 버퍼를 이용하지 않는다는 점을 감안하면 1D

표 4.7 더미 비트 재사용 방법의 성능

Encoding scheme	Dummy bit / length header	No dummy bit / length header	No dummy bit / no length header (optimal)
PSNR (dB)	43.41	44.03	44.30

SPIHT 알고리즘도 충분히 좋은 압축 효율을 보여주는 것으로 생각할 수 있다. 따라서 저비용 고효율 압축 시스템에서 1D 알고리즘이 훌륭한 옵션이 될 수 있다.

다음으로는 4.1.2장, 4.2.1장, 4.2.2장에서 제안한 압축 효율 향상 방법의 성능을 구체적으로 확인하도록 한다. 표 4.7은 더미 비트 재사용 방법의 성능을 보인다. 실험을 위해 24개의 Kodak 이미지를 이용하였고, 평균 PSNR을 표기하였다. 표 4.7의 모든 결과는 소프트웨어 시뮬레이션으로 얻어진 결과이다. 참조 소프트웨어는 하드웨어 동작을 시뮬레이션하기 위해 작성되었다. 표 4.7의 두 번째 열에서 보인 첫 번째 결과는 더미 비트 재사용 방법을 적용하지 않았을 때의 결과이다. 다시 말해, 모든 더미 비트 공간이 '0'으로 채워졌을 때의 결과이다. 또한 첫 번째 결과에는 크기 비트와 부호 비트의 경계를 디코더에 분명히 전달하기 위한 7 비트 크기의 헤더 비트가 들어간 결과이다. 이때의 평균 PSNR은 43.41 dB로 확인되었다. 표 4.7의 세 번째 열에서 확인 가능한 두 번째 결과는 더미 비트 재사용 방식을 이용했을 때의 결과이다. 하지만 7 비트 크기의 헤더는 계속 사용되는 상황이다. 두 번째 결과의 경우 0.62 dB의 PSNR 향상을 보인다. 이와 같은 성능 향상은 제안한 더미 비트 재사용 방법에 기인한 것이라 생각할 수 있다. 참고로 이 결과는 표 4.5의 네 번째 행에서도 확인 가능하다. 표 4.5의 마지막 열에서 확인 가능한 세 번째 결과는 7 비트

표 4.8 더미 비트 횟수에 따른 성능

Number of dummy bits	0	10	20	40 (proposed)	Infinite
PSNR (dB)	43.41	43.74	43.95	44.03	44.03

표 4.9 절반-패스 처리 방법의 성능

Pass size	Original	1/2 of original (proposed)	1/4 of original	Optimal
PSNR (dB)	44.12	44.22	44.27	44.30

크기의 헤더까지 제거된 상황의 결과이다. 이 경우 헤더는 없지만 크기 비트와 부호 비트가 겹치는 상황이 발생하지 않는다고 가정한다. 다시 말해, 이상적인 경우의 결과이다. 이 경우 평균 PSNR은 0.27 dB만큼 향상되었다.

다음 실험은 더미 비트 재사용 방법에서 더미 비트 재사용 횟수에 따른 압축 효율을 확인하기 위한 실험이다. 더미 비트 재사용 횟수는 더미 비트를 저장하는 버퍼의 크기에 영향을 미친다. 실험을 위해 24개의 Kodak 이미지를 이용하였고, 평균 PSNR 수치를 표 4.8에 표기하였다. 표 4.8에서 보인 바와 같이 더미 비트 재사용 횟수가 늘어날수록 PSNR 역시 향상된다. 그러나 더미 비트 재사용 횟수가 40에 이르면 성능 향상이 포화 상태가 된다. 따라서 본 논문에서는 더미 비트 재사용 횟수를 40으로 정하도록 한다. 단, 흑백 이미지의 경우 20으로 정한다.

표 4.9는 4.2.2장에서 제안한 절반-패스 처리 방법의 성능을 보인다. 이전 실험들과 마찬가지로 24개의 Kodak 이미지가 실험에 이용되었고, 평균 PSNR이 표 4.9에 표기되었다. 표 4.9의 두 번째 열은 그림 4.4 (a)에

표 4.10 분류 비트 재배치 방법의 성능

Compression ratio	3	4	5	6
Average PSNR (dB) of the original order	44.22	39.48	36.10	33.78
Average PSNR (dB) of the proposed order	44.37	39.75	36.48	34.20
Difference	0.15	0.27	0.38	0.42

서 보인 원래의 패스 크기를 이용했을 때의 평균 PSNR을 보인다. 반면에 세 번째 열은 그림 4.4 (b)에서 보인 절반-패스 크기를 이용했을 때의 평균 PSNR을 보인다. 절반-패스 크기를 이용할 경우 0.1 dB의 평균 PSNR이 향상된다. 참고로 이 결과는 표 4.5의 다섯 번째 행에서도 확인 가능하다. 표 4.9의 네 번째 열은 1/4 크기로 나눠진 패스에 대한 PSNR 결과를 보인다. 이 경우 평균 PSNR이 0.05 dB만큼 추가로 향상되었다. 마지막 열은 이상적인 상황에서의 평균 PSNR을 보인다. 이 경우 0.03 dB만큼의 PSNR이 추가적으로 향상된다. 패스가 더 작은 단위로 나눠질수록 이상적인 상황의 결과에 가까워진다. 그러나 하드웨어 크기는 증가하므로 본 논문에서는 절반-패스 크기를 하드웨어 구현에 적용하도록 한다.

표 4.10은 4.1.2장에서 제안한 분류 비트 재배치 방법의 성능을 보인다. 원래의 방법에서는 RP, SP, FRP 순서로 비트가 저장된다. 제안 비트스트림 포맷에서는 SP에서 생성된 비트스트림이 RP, FRP에서 생성된 비트스트림 이후에 저장이 된다. 실험을 위해 24개의 Kodak 이미지가 이용되었고, 평균 PSNR이 표 4.10에 표기되었다. 표 4.10의 두 번째 행은 원래의 순서가 적용되었을 때의 결과를 보인다. 반면에 세 번째 행은 제안 순서가 적용되었을 때의 결과를 보인다. 압축률에 따라 비트스트림에 포함되는 분류 비트의 비율이 달라지므로 목표 압축률을 3, 4, 5, 6으로 설정하

여 실험하였다. 제안 방법은 기존 방법에 비해 항상 높은 성능을 보인다. 목표 압축률이 3인 경우 평균 PSNR은 0.15 dB만큼 향상된다. 성능은 압축률이 증가될수록 더 크게 향상된다. 목표 압축률이 6인 경우 평균 PSNR은 0.42dB만큼 향상된다.

제 5 장 고충실도 RGBW 컬러 이미지 압축을 위한 고정 압축비 VLC

5.1 제안 알고리즘

이 장은 RGBW 컬러 이미지 압축을 위한 알고리즘을 제안하도록 한다. 대상 시스템의 목표 조건은 표 5.1에서 확인할 수 있다. 우선 제안 알고리즘은 디스플레이에 실제로 출력되는 이미지를 압축한다. 따라서 근접-무손실 압축을 목표로 한다. 그리고 압축 이득을 위해서는 압축 방식이 가벼워야 하므로 인트라 정보로만 코딩하도록 한다. 제안 알고리즘은 10 비트-너비의 RGBW 데이터를 압축한다. 이때, 고충실도 특성을 위하여 [17-18]과 같이 컬러 변환과 컬러 성분 서브-샘플링을 수행하지 않는다. 본 시스템에서 목표로 하는 압축률은 2.5이다. 예를 들어 1×32 크기의 블록(1,280 비트)은 512 비트로 압축이 된다. 제안 알고리즘은 4K 이미지를 60 fps로 처리할 수 있는 속도를 목표로 한다. 따라서 가벼우면서 압축 효

표 5.1 목표 시스템

Attribute	Value
Image quality	Visually lossless
Coding across frames	No, intra only
Color component type	RGBW
Component bit-width	10
Sub-sampling	No, 4:4:4:4
Compression ratio	2.5
Throughput	4K (3840×2160) @60fps
Line buffer	1-line

율이 높은 방식이 본 시스템에 적합하다. 그리고 라인 버퍼의 경우 인코더와 디코더가 각각 1-라인 크기의 메모리를 이용하도록 한다. 이어지는 내용은 RGBW 인터-컬러 연관성을 이용한 예측 방법과 VLC의 압축 효율을 유지하면서 고정 압축비를 맞출 수 있는 방법을 제안하도록 한다.

5.1.1 RGBW 인터-컬러 연관성을 이용한 예측 방식

예측 기반 압축 알고리즘의 성능은 크게 두 부분에서 결정이 된다. 첫 번째는 잔차(또는 차분값)의 크기를 감소시키는 예측 부분이다. 그리고 두 번째는 잔차를 실제로 코딩하는 부분이다. 이 장에서는 RGBW 도메인에 존재하는 인터-컬러 연관성을 이용하여 잔차를 감소시키는 예측 방법(inter-color correlation-based prediction; ICP)을 제안하도록 한다.

제안하는 예측 방식은 두 단계의 차분 과정으로 구성된다. 첫 번째 차분은 공간적 연관성을 이용하여 잔차를 계산하는 단계이다. 작은 하드웨어 크기를 위해 제안 알고리즘은 1-라인 크기의 버퍼를 이용한다. 그리고 실시간 하드웨어 처리를 위해 현재 픽셀에서 멀리 떨어진 픽셀을 참조하지 않는다. 따라서 이용할 수 있는 공간적 연관성이 크지 않다. 본 연구에서는 복잡도 대비 우수한 성능을 보이는 MED 예측기[8]를 이용하여 현재 픽셀에 대한 1차 잔차를 계산하도록 한다.

1차 차분에 의해 생성된 잔차는 2차 차분을 통해 그 크기를 더 감소시키게 된다. 2차 차분은 공간적 연관성이 아닌 인터-컬러 연관성을 이용하여 잔차를 계산한다. 따라서 2차 차분은 주변 픽셀과는 무관하게 현재 픽셀 내부에서만 계산을 수행한다. 제안하는 2차 차분 방식은 R, G, B, W 서브-픽셀들의 근사 평균값을 이용하여 각 컬러 성분의 잔차를 감소시킨

다. 이와 같은 방식은 인터-컬러 연관성이 낮은 RGBW 컬러 도메인에서 상대적으로 더 높은 잔차 감소 효율을 보인다.

2차 차분을 수행하기 위해서는 차분 기준으로 삼을 참조값이 필요하다. 2차 차분의 참조값을 계산하는 방법은 다음과 같다. 우선 1차 차분 결과인 Res_R, Res_G, Res_B, Res_W가 2차 차분의 입력값으로 들어온다. 이 값들은 식 (5.1)을 통해 참조값 x를 생성한다. 평균의 의미로는 4로 나누는 것이 적합하나 이 경우 디코더에서 x를 유추할 수 없게 된다. 따라서 4가 아닌 가까운 수인 3을 이용하도록 한다.

$$x = \left\lfloor \frac{Res_R + Res_G + Res_B + Res_W}{3} \right\rfloor \tag{5.1}$$

참조값 x가 계산되면 식 (5.2)를 통해 식 (5.1)의 내림 과정으로 버려지는 수를 계산하도록 한다. 단, 식 (5.2)의 결과값인 신택스가 음수인 경우 이 값을 양수로 만들기 위해 몫에 해당하는 식 (5.1)의 참조값 x에서 1을 뺀다. 그리고 나머지에 해당하는 신택스에 3을 더하도록 한다. 가령, -5/3이 '-1-2/3'으로 표현이 되지만 '-2+1/3'으로도 표현이 가능한 특성을 이용한 것이다. 식 (5.2)의 신택스 결과는 2 비트로 코딩되며, 이때 가능한 값은 0, 1, 2이다. 식 (5.1)에서 5가 아닌 3을 나눗셈에 이용한 이유는 신택스에 할당되는 비트 수를 제한시키기 위해서이다. 5를 이용하는 경우 가능한 나머지가 0, 1, 2, 3, 4이므로 신택스에 할당되는 비트 수가 한 비트 더 늘어나는 문제가 있다.

$$syntax = (Res_R + Res_G + Res_B + Res_W) \,\%\, 3 \tag{5.2}$$

식 (5.1)을 통해 계산된 참조값 x 는 1차 차분 결과인 Res_R, Res_G, Res_B, Res_W의 참조값으로 이용된다. 식 (5.3)은 2차 차분 과정을 보인다. 식 (5.3)

에서 Res'_{color}는 해당 컬러 성분에서의 2차 잔차를 의미한다. 식 (5.3)으로 계산된 2차 잔차는 인코더의 입력값으로 이용되며, 인코딩을 통해 생성된 비트는 식 (5.2)의 신택스 비트와 함께 최종 비트스트림으로 출력된다.

$$Res'_{color} = Res_{color} - x \tag{5.3}$$

디코더는 인코더의 역방향 계산을 수행한다. 우선 디코딩을 통해 2차 잔차인 Res_R', Res_G', Res_B', Res_W' 를 복원한다. 그런 다음 복원된 2차 잔차를 모두 더하도록 한다. 식 (5.3)를 이용하면 2차 잔차의 합은 1차 잔차의 합 형태로 표현이 가능하다. 이때, 1차 잔차의 합은 3n, 3n+1, 3n+2 (n=0, 1, ...) 형태 여부에 따라 참조값 x와 신택스가 달라진다. 가령, 1차 잔차의 합이 3n이라면, 식 (5.1)과 식 (5.2)를 통해 참조값 x는 n, 신택스는 0이 된다. 따라서 1차 잔차의 합은 '3x+신택스'로 치환할 수 있다. 1차 잔차의 합이 3n+1이라면, 참조값 x는 n, 신택스는 1이 된다. 따라서 3n의 경우와 마찬가지로 1차 잔차는 '3x+신택스'로 치환 가능하다. 1차 잔차의 합이 3n+2인 경우에도 '3x+신택스'로 치환 가능하다. 따라서 2차 잔차의 합은 식 (5.4)와 같이 표현할 수 있다.

$$\begin{aligned} Res'_R + Res'_G + Res'_B + Res'_W \\ &= Res_R + Res_G + Res_B + Res_W - 4x \\ &= syntax - x \end{aligned} \tag{5.4}$$

식 (5.4)에서 참조값 x를 좌항으로 이동시키고 2차 잔차의 합을 우항으로 이동시키면 식 (5.5)의 결과를 얻을 수 있다. 식 (5.5)을 통해 전달받은 신택스와 2차 잔차를 이용하여 디코더가 인코더에서 이용한 참조값을 정확하게 계산할 수 있음을 알 수 있다.

$$x = syntax - (Res'_R + Res'_G + Res'_B + Res'_W) \tag{5.5}$$

디코더에서 계산된 참조값 x를 이용하여 식 (5.6)과 같이 1차 잔차를 복원하도록 한다. 식 (5.6)은 식 (5.3)의 역연산이다. 복원된 1차 차분 결과는 MED 예측기의 역연산을 통해 최종 픽셀값으로 복원이 된다.

$$Res_{color} = Res'_{color} + x \tag{5.6}$$

5.1.2 고정 압축비를 위한 Golomb-Rice 코딩

이 장에서는 예측 기반 압축 방식의 성능을 결정하는 두 번째 요소인 코딩 방식에 대해 다루도록 한다. 일반적인 VLC 방법들은 고정 압축비에 취약한 특성을 보인다. 본 논문에서는 하드웨어 구현에 유리하면서 압축 성능이 우수한 것으로 알려진 Golomb-Rice 코딩을 기반으로 한 고정 길이 압축 방법을 제안하도록 한다.

본 연구에서 제안하는 방식은 그림 5.1과 같이 프리-코더, 포스트-코더 등 두 개의 인코더를 이용하여 압축을 수행한다. 프리-코더는 특정 상황에 대하여 실제 인코딩을 진행한 후 다른 모든 상황에 대한 예측 인코딩 정보를 계산하여 포스트-코더에 전달할 QL을 선택한다. 가령, 양자화 파라미터 QL이 0인 상황에 대해 실제 인코딩을 수행하고, 다른 QL이 이용될 때 몇 비트가 생성될 것인지를 예측한다. 그런 다음 목표 비트스트림 길이를 넘지 않는 최적의 QL을 선택하여 포스트-코더에 전달한다. 포스트-코더는 프리-코더로부터 전달받은 QL을 이용하여 실제 비트스트림을 생성한다. 프리-코더와 포스터-코더 사이에는 고정 비트스트림 길이를 갖고 동일한 QL을 적용하는 코딩 블록 크기만큼의 딜레이가 발생하기 때문에 해당 데이터를 저장하기 위한 버퍼 메모리가 필요하다.

그림 5.1의 구조를 실제로 적용하기 위해서는 프리-코더가 제한된 실

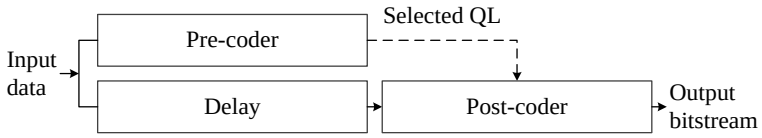


그림 5.1 제안하는 인코더 구조

제 인코딩 정보를 가지고 다른 예측 인코딩 정보들을 계산할 수 있어야 한다. 제안하는 알고리즘에서 이용하는 Golomb-Rice 코더는 나눗셈의 몫과 나머지를 코딩하므로 제수와 피제수만 알고 있으면 생성되는 비트스트림의 길이를 쉽게 예측할 수 있다. 예를 들어 제수가 고정되어 있는 경우, 피제수는 QL이 1씩 증가할 때마다 그 값이 거의 절반이 되는 관계를 갖는다. 따라서 특정 QL에 대한 비트스트림의 길이가 주어진다면 다른 QL에 대한 비트스트림의 길이를 예측할 수 있다. 이와 같이 예측된 비트스트림의 길이를 토대로 목표 비트스트림 길이를 넘는 경우를 사전에 방지할 수 있다.

표 5.2는 특정 QL에 대한 인코딩 결과를 이용하여 다른 QL에 대한 인코딩 결과를 예측한 한 예를 보인다. 간단한 설명을 위해 1차 차분은 MED가 아닌 DPCM을 이용하는 것으로 가정하도록 한다. 이때, -1 번째 픽셀값은 (440, 218, 428, 409)이며, 제수는 1로 고정되어 있다. 표 5.2에서는 QL이 0일 때 실제 인코딩이 진행되었고, QL이 2인 경우의 결과는 QL이 0일 때의 결과를 이용하여 예측되었다. 이때, 제수가 1로 고정되었으므로 Golomb-Rice 코딩의 나머지 길이 역시 고정된다. 따라서 해당 블록의 비트스트림 길이는 몫의 길이에 의해 결정이 된다. QL이 0에서 2로 증가하면, Golomb-Rice 인코더의 입력 데이터 값이 1/4배가 된다. 이때, 입력 데이터는 식 (2.6)에서 보인 ‘source’이다. 제수가 고정되어 있으므로

표 5.2 인코딩 길이 예측의 한 예

		Real calculation (QL = 0)				Real calculation (QL = 2)				Prediction (QL = 2)			
Pixel no.		0	1	2	3	0	1	2	3	0	1	2	3
Original data	R	411	462	534	538	102	115	133	134	-	-	-	-
	G	167	202	274	309	41	50	68	77	-	-	-	-
	B	350	409	530	583	87	102	132	145	-	-	-	-
	W	315	380	514	579	78	95	128	144	-	-	-	-
Res	R	-29	51	72	4	-8	13	18	1	-	-	-	-
	G	-51	35	72	35	-13	9	18	9	-	-	-	-
	B	-78	59	121	53	-20	15	30	13	-	-	-	-
	W	-94	65	134	65	-24	17	33	16	-	-	-	-
Res’	R	55	-19	-61	-48	14	-5	-15	-12	-	-	-	-
	G	33	-35	-61	-17	9	-9	-15	-4	-	-	-	-
	B	6	-11	-12	1	2	-3	-3	0	-	-	-	-
	W	-10	-5	1	13	-2	-1	0	3	-	-	-	-
Source	R	110	37	121	95	28	9	29	23	-	-	-	-
	G	66	69	121	33	18	17	29	7	-	-	-	-
	B	12	21	23	2	4	5	5	0	-	-	-	-
	W	19	9	2	26	3	1	0	6	-	-	-	-
Len. of quotient	R	55	18	60	47	14	4	14	11	13	4	15	11
	G	33	34	60	16	9	8	14	3	8	8	15	4
	B	6	10	11	1	2	2	2	0	1	2	2	0
	W	9	4	1	13	1	0	0	3	2	1	0	3
Total length		410				119				121			

피제수가 1/4배가 되면 몫의 크기도 1/4배가 된다. Golomb-Rice 코딩에서 픽셀당 몫에 할당되는 비트의 수는 ‘몫의 크기 + 1’이다. 따라서 표 5.2의 마지막 네 열과 같이 QL이 2인 경우를 실제로 코딩하지 않더라도 생성되는 비트스트림의 길이를 간단하게 예측할 수 있다. QL이 n만큼 증가하였을 때의 일반화된 픽셀당 예측 비트스트림 길이는 식 (5.7)과 같다.

$$Predicted\ length_{GR} = \left\lfloor \left\lfloor source/2^{k} \right\rfloor / 2^{n} \right\rfloor + k + 1 \qquad (5.7)$$

이와 같이 제안한 방식은 특정 상황에 대한 인코딩 정보를 이용하여 다른 모든 상황에 대한 인코딩 결과를 예측한다. 그러나 실제 비트스트림과 예측 비트스트림의 길이는 차이가 날 수 있다. 가령, 표 5.2의 경우 실

제 길이는 119이지만 예측 길이는 121로 2만큼 차이가 난다.

인코딩된 비트스트림의 길이와 예측 비트스트림의 길이에서 차이가 나는 이유는 양자화 과정이 적용되는 시점이 다르기 때문이다. 실제 비트스트림에서는 1차 차분과 2차 차분을 수행하기 전에 이미 양자화 과정이 적용되어 있다. 그러나 예측 비트스트림에서는 식 (5.7)과 같이 마지막에 양자화 과정을 적용한다. 가령, 표 5.2의 0 번째 픽셀, R 성분의 경우 실제 계산에서는 원본 픽셀 단계($\lfloor 411/4 \rfloor = 102$)에서 소수점 버림이 발생한다. 이때 발생한 에러는 이후의 계산에 계속 영향을 미치게 된다. 반면에 예측 계산에서는 이미 계산된 QL 0의 몫으로 QL 2의 몫을 예측($\lfloor 55/4 \rfloor = 13$)할 때 소수점 버림이 발생한다. 그 결과 실제 몫의 길이는 14이지만 예측 몫의 길이가 13이 되어 1만큼의 차이가 발생하게 된다.

인코딩된 비트스트림의 길이가 목표 비트스트림 길이를 넘지 않기 위해서는 예측 몫이 실제 몫보다 무조건 커야 한다. 이와 같은 상황을 보장하기 위하여 예측 몫에 보상값을 더하도록 한다. 그러나 픽셀마다 항상 충분한 보상값이 주어질 경우 예측 비트스트림의 길이가 실제 비트스트림의 길이보다 훨씬 크게 되어 이상적인 QL보다 더 큰 QL이 선택되는 문제가 발생한다. 이와 같이 크게 선택된 QL은 압축 효율을 감소시킨다.

보상값 이용을 피하기 위하여 다음과 같은 방법들을 이용할 수 있다. 첫 번째 방법은 예측값 이용 빈도를 낮추는 방법이다. 목표 압축률이 2.5인 경우 실험적으로 QL 2가 가장 많이 선택된다. 따라서 프리-코더에서 이용하는 초기 QL을 0이 아닌 2로 설정하면, 예측값이 아닌 실제값을 이용하는 빈도수가 증가하므로 보상값 이용을 피할 수 있다. 두 번째 방법은 보상값을 주지 않는 회로를 기존 회로와 병렬로 동작시키는 방법이다.

만약 보상값을 주지 않는 회로에서 생성된 비트스트림의 길이가 목표 비트스트림 길이보다 작을 경우 이 비트스트림이 최종 비트스트림으로 출력된다. 이 경우 보상값 이용 빈도를 상당수 감소시킬 수 있다. 그러나 두 번째 방법은 보상값을 이용하는 포스트-코더와 이용하지 않는 포스트-코더가 함께 존재하여야 하므로 하드웨어 비용이 증가한다.

5.1.3 알고리즘 요약

그림 5.2는 5.1.1장과 5.1.2장에서 제안한 RGBW 인터-컬러 연관성을 이용한 예측 방식과 고정 압축비를 위한 Golomb-Rice 코딩 등을 적용한 전체 인코딩 알고리즘의 순서도를 보인다. 우선 인코더의 입력값으로는 10 비트-너비 RGBW 데이터가 들어온다. 5.1.2장에서 설명한 바와 같이 인코더의 프리-코더는 들어온 픽셀들에 대하여 QL 2의 양자화 과정을 수행한다. 그런 다음, 1차 차분과 2차 차분을 수행한다. 이와 같은 작업을 미리 정한 코딩 블록 크기만큼 수행하도록 하며, 계산된 비트스트림 길이 정보는 버퍼 메모리에 잠시 저장하도록 한다. 코딩 블록 크기만큼의 결과가 모이게 되면 해당 정보를 기반으로 QL을 결정하도록 한다. 프리-코더에서 결정한 QL 정보는 포스트-코더로 전달되어 실제 비트스트림 생성에 이용된다. 포스트-코더는 프리-코더가 연산을 시작하고 한 코딩 블록이 지난 시점에 연산을 시작하며, 잔차를 계산한 다음 Golomb-Rice 인코더를 이용하여 최종 비트스트림을 생성한다.

생성된 비트스트림은 프레임 메모리에 저장되었다가 디코더에 전달된다. 디코더의 연산 과정은 인코더의 연산 과정과 반대이다. 즉, Golomb-Rice 디코딩, 1차 잔차 복원(2차 차분 역연산), 픽셀값 복원(1차 차분 역연

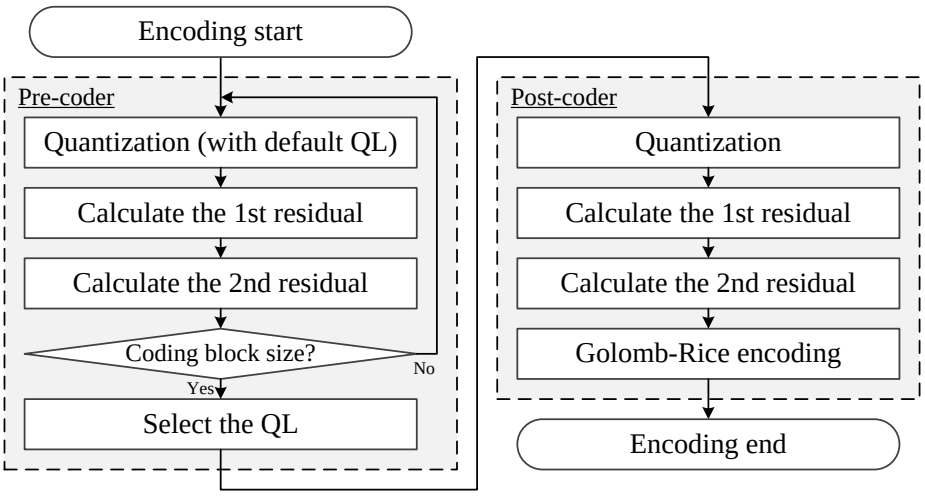


그림 5.2 제안하는 인코더 알고리즘의 순서도

산) 등으로 구성된다. 그러나 디코더는 전달받은 QL을 이용하기 때문에 QL을 계산하기 위한 연산이 요구되지 않는다는 점에서 인코더와 차이가 난다.

5.2 하드웨어 구조

이 장은 제안 알고리즘에 대한 하드웨어 구조를 보이도록 한다. 제안 하드웨어 구조는 실시간으로 동작한다. 표 5.1에서 보인 고속 조건을 만족시키기 위해 인코더의 입력 데이터와 디코더의 출력 데이터는 매 사이클당 4 픽셀씩 연속적으로 전송된다. 한 픽셀은 10 비트-너비 크기의 R, G, B, W 서브-픽셀로 구성되어 있다. 따라서 매 사이클당 160 비트의 데이터가 전송된다. 하드웨어에서의 코딩 블록 크기는 1×32로 설정하였으며, Golomb-Rice 코더의 제수 k는 1로 고정되었다. 목표 압축률은 2.5이다.

5.2.1 인코더 구조

제안하는 인코더 하드웨어 구조는 그림 5.3과 같다. 전체적으로 살펴보면 QL을 결정하는 프리-코더, 실제로 인코딩을 수행하는 포스트-코더, 프리-코더와 포스트-코더 사이의 딜레이를 지원하기 위한 딜레이 버퍼, 윗 라인 정보를 저장하는 라인 버퍼 등으로 구성되어 있다. 인코더에는 10 비트-너비, 4 픽셀, 4 컬러에 해당하는 160 비트 데이터가 매 사이클마다 연속적으로 들어온다. 제안하는 압축 방식은 1차 차분 단계에서 윗 라인의 픽셀들을 이용하므로 컨택스트 생성기 모듈이 라인 버퍼로부터 위쪽 4 픽셀을 입력 데이터와 같은 시간대에 읽어온다. 따라서 매 사이클마다 인코더에 들어오는 데이터는 총 320 비트이다. 컨택스트 생성기 모듈에 모인 320 비트는 프리-코더와 딜레이 버퍼에 각각 전달이 된다. 딜레이 버퍼의 경우 단일-포트 SRAM 으로 구성되고, 1 코딩 블록 크기를 갖는다.

프리-코더의 잔차 계산 모듈은 크게 두 단계로 구성된다. 기능상으로는 1차 잔차 계산과 2차 잔차 계산으로 단계를 나누는 것이 옳으나 1차 차분 과정과 2차 차분 과정의 계산량 균형이 맞지 않는 문제가 있다. 따라서 식 (5.1)과 (5.2)의 1차 차분 덧셈 연산을 1차 차분으로 옮겨 ‘잔차 계산0’ 단계를 형성하고, 나머지 2차 차분 과정을 ‘잔차 계산1’ 단계로 구성하도록 한다.

QL 선택 모듈은 1 코딩 블록 크기만큼 모인 잔차 결과를 이용하여 목표 압축률을 만족시킬 수 있는 최적의 QL을 선택한다. 비트-플레인의 수가 10이므로 QL을 알려주기 위한 신호의 비트-너비는 4이다. 그림 5.4은 QL 선택 모듈의 블록 다이어그램을 보인다. 그림 5.4 (a)는 QL 3에 대

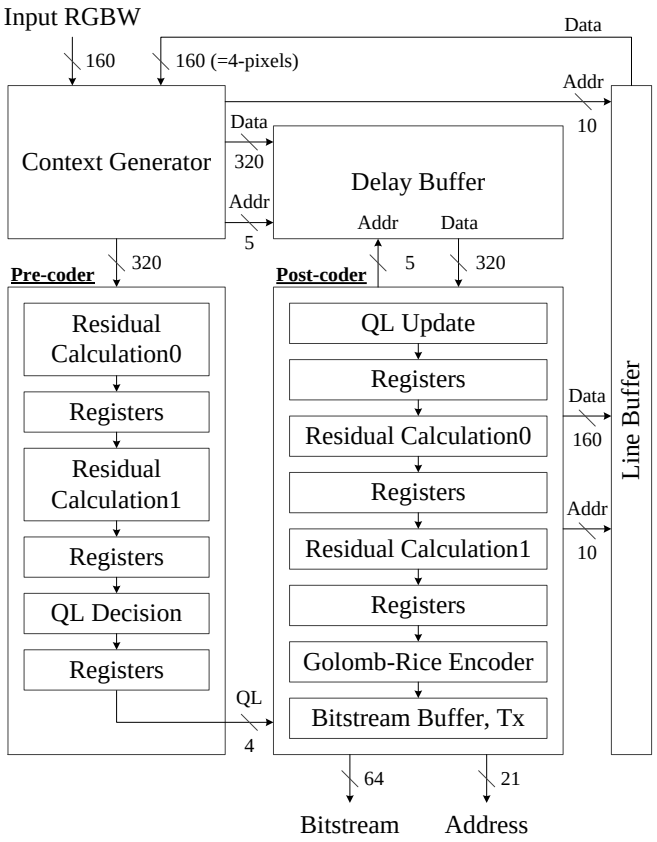


그림 5.3 제안하는 인코더 하드웨어의 구조

한 서브-몫의 합 계산 회로를 보인다. 실제로는 4 픽셀의 몫이 동시에 더해지지만 공간 제약상 한 픽셀의 서브-몫의 합 계산 회로만을 보이도록 한다. 앞서 설명한 내용과 같이 Golomb-Rice 코더에서 생성된 비트스트림 중 길이가 가변적인 부분은 단일 코드로 코딩되는 몫 부분이다. 따라서 QL 선택 모듈은 몫만을 고려한다. 식 (2.6)에서 잔차의 부호를 제거하기 위해 2를 곱하므로 제수가 1인 경우 몫의 길이는 2차 잔차의 절대값으로 생각할 수 있다. 따라서 몫의 길이 대신 2차 잔차를 이용하더라도 해석에 무리가 없다. 단, 2차 잔차가 음수인 경우 2차 잔차의 절대값과 몫을 동일

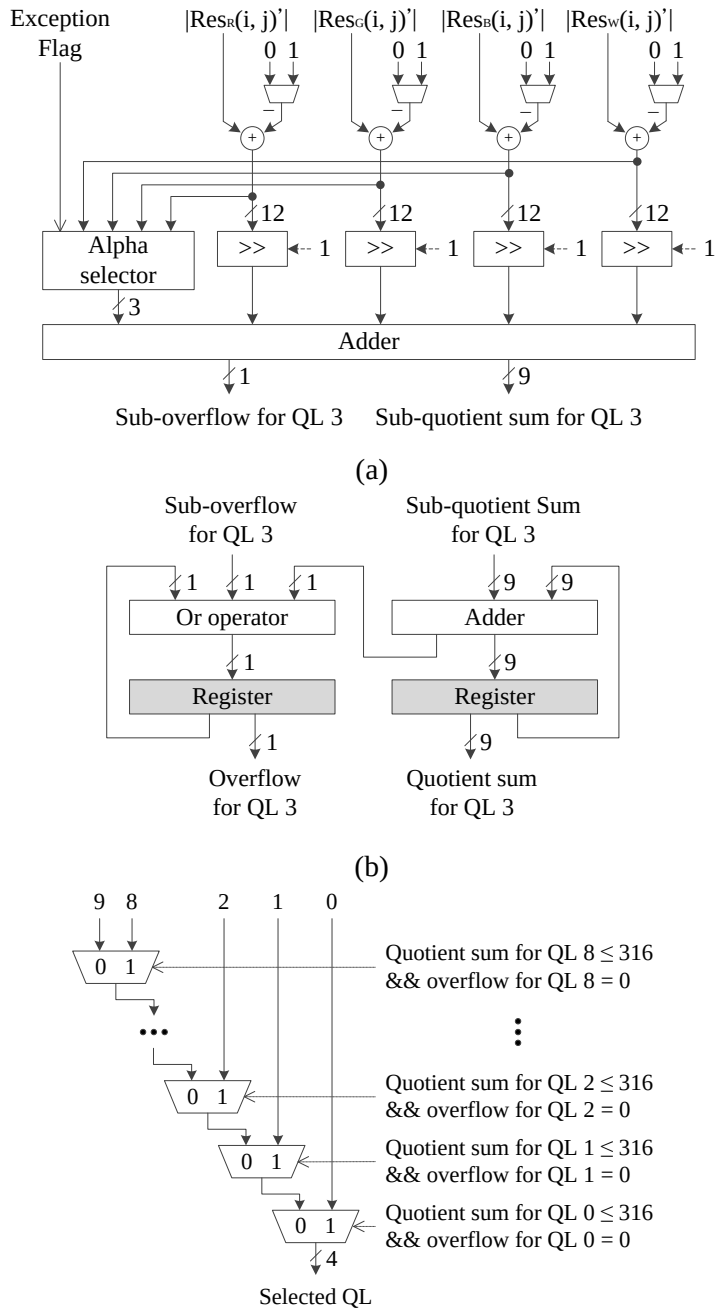


그림 5.4 QL 선택 모듈의 구조. (a) QL 3에 대한 서브-몫의 합 계산 (b) QL 3에 대한 최종 몫의 합 계산 (c) QL 선택

하게 만들기 위하여 2차 잔차의 절대값에 1을 빼도록 한다. 변형된 2차 잔차는 보상값과 더해져 QL 3에 대한 서브-몫의 합을 계산한다. 예외 플래그는 원래의 현재 픽셀값이 참조 픽셀값과 동일할 경우 보상값을 0으로 만드는 기능을 수행한다. 연산자의 비트-너비를 제한시키기 위하여 QL 3에 대한 서브-몫의 합이 이미 목표 비트스트림 길이를 넘어가는 경우 오버플로 비트 1을 출력하도록 한다. 그림 5.4 (b)와 같이 매 사이클마다 계산되는 서브-몫의 합은 8 사이클 동안 누적되어 최종 몫의 합을 계산한다. 오버플로 플래그 역시 8 사이클 동안 누적된다. 각 QL에 대해 계산된 최종 몫의 합은 그림 5.4 (c)와 같이 QL 0에 대한 최종 몫의 합부터 오름차순으로 몫에 할당된 목표 비트 길이(316)를 만족하는지 확인하게 된다. 이때, 해당 QL에 대한 오버플로 비트가 1인 경우 무조건 현재 QL은 선택되지 않으며, 다음 QL 확인 과정으로 넘어가게 된다. 마지막 조건까지 만족되지 않은 경우 QL 9가 최종 QL로 선택된다.

포스트-코더는 1 코딩 블록에 대한 프리-코더의 연산이 완료가 되면 매 사이클마다 320 비트를 전달받아 실제 인코딩 과정을 수행한다. 포스트-코더에 전달된 입력 데이터는 QL 업데이트 모듈에서 프리-코더로부터 전달받은 QL에 따라 양자화된다. 양자화된 현재 픽셀 데이터는 다음 라인 코딩을 위해 다시 업-스케일링되어 라인 버퍼에 저장된다. 라인 버퍼의 경우 읽고, 쓰기가 동시에 진행되기 때문에 이중-포트 SRAM으로 구성된다. 양자화된 데이터는 잔차 계산 모듈에서 1차 잔차와 2차 잔차를 계산한다. 2차 잔차가 계산되면 실제 Golomb-Rice 인코딩을 수행한다. 생성된 비트스트림은 패킹 과정을 거친 후 비트스트림 버퍼에 잠시 저장이 되었다가 매 사이클당 64 비트의 속도로 프레임 메모리에 전달된다.

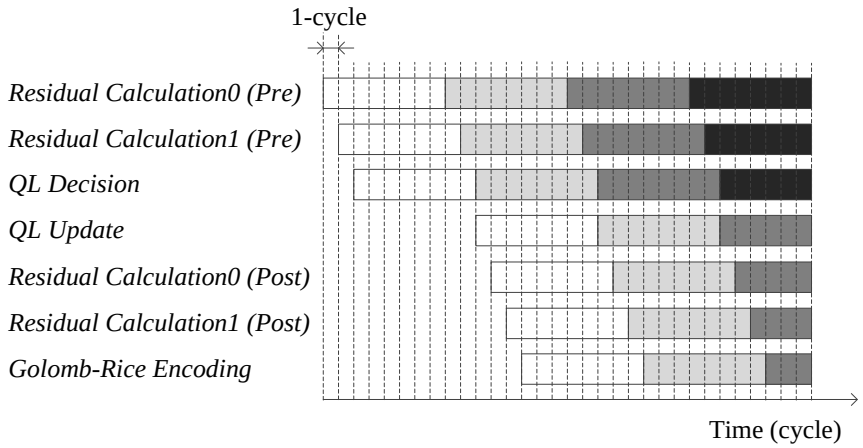


그림 5.5 제안 인코더의 타이밍 다이어그램

제안하는 인코더 하드웨어의 타이밍 다이어그램은 그림 5.5와 같다. 이때, 가로축은 시간을 의미하고, 한 칸은 1 사이클을 의미한다. 그리고 서로 다른 색의 사각형은 서로 다른 1×32 코딩 블록을 의미한다. 인코더의 입력 데이터가 매 사이클마다 4 픽셀씩 들어오므로 8 사이클을 간격으로 처리하는 블록이 바뀌게 된다. 그림 5.1에서 보인 바와 같이 프리-코더와 포스트-코더 사이에는 1 코딩 블록만큼의 딜레이가 발생한다. 따라서 프리-코더의 QL 선택 모듈과 포스트-코더의 QL 업데이트 모듈 사이에 1 블록만큼의 지연시간이 발생한다.

5.2.2 디코더 구조

제안하는 디코더 하드웨어 구조는 그림 5.6과 같다. 우선 프레임 메모리로부터 입력 비트스트림을 받아오기 위하여 프레임 메모리 주소 생성기가 프레임 메모리 주소값을 계산한다. 계산된 주소값에 해당하는 입력 비트스트림은 매 사이클마다 64 비트씩 디코더에 전달이 된다.

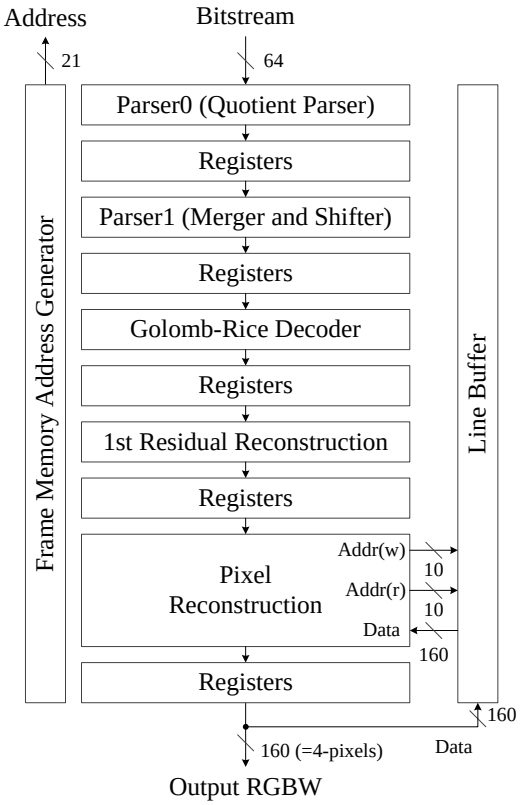


그림 5.6 제안하는 디코더 하드웨어의 구조

디코더 구조의 경우 비트스트림 포맷이 전체 구조에 큰 영향을 미친다. 블록의 크기가 1×32이고, 압축률이 2.5인 경우 Golomb-Rice 인코더에서 생성된 1 코딩 블록에 대한 비트스트림은 4 비트의 QL, 64 비트의 2차 차분 신택스, 316 비트의 몫, 128 비트의 나머지 등 총 512 비트로 구성되어 있다. QL은 비트-플레인의 수가 10이므로 4 비트로 전달되고, 2차 차분 신택스는 식 (5.5)을 수행하기 위해 각 픽셀마다 2 비트씩 출력된다. 따라서 그 크기가 64 비트이다. 몫과 나머지는 각각 총 128(=1×32×4) 개의 서브-픽셀 정보를 포함한다. 나머지의 경우 제수 k가 1로 고정되어 있

QL	Quotient	Remainder	Syntax
4	36	16	8

(a)

Quotient	Remainder	Syntax
40	16	8

(b)

그림 5.7 사이클에 따른 비트스트림 구조.

(a) 사이클=8n (b) 사이클≠8n

으므로 항상 128 비트가 출력된다. 몫은 남은 비트스트림의 공간을 이용한다. 따라서 몫에 할당된 비트 수는 316이다. 그림 5.7은 매 사이클마다 생성되는 비트스트림의 포맷을 보인다. 효과적인 디코더 구조를 위하여 몫, 나머지, 2차 차분 신택스 등을 8 사이클 동안 나눠 전송한다. 가령, 1 코딩 블록에 대한 나머지의 경우 총 128 비트이므로 한 사이클당 16 비트씩 전송된다. 그러나 각 코딩 블록에 대한 첫 번째 사이클의 경우 그림 5.7 (a)와 같이 4 비트의 QL 정보가 포함된다. 따라서 몫을 36 비트만 보내게 한다. 나머지 사이클의 경우 그림 5.7 (b)와 같이 64 비트에 몫, 나머지, 2차 차분 신택스를 각각 40 비트, 16 비트, 8 비트씩 포함시켜 전송하게 한다.

그림 5.6의 파서 모듈에 전달된 비트스트림은 몫, QL, 나머지, 2차 차분 신택스 정보 등으로 파싱된다. 몫을 제외한 나머지 성분은 그 길이를 쉽게 계산할 수 있기 때문에 파싱이 어렵지 않다. 그러나 316 비트에 해당하는 몫은 비트스트림 내부에 존재하는 의존 관계로 인해 파싱을 쉽게 수행할 수 없다. 이 의존 관계는 디코더의 고속 동작을 방해하는 가장 큰 요인이다. 그림 5.8은 해당 의존 관계의 한 예를 보인다. 그림 5.8의 316

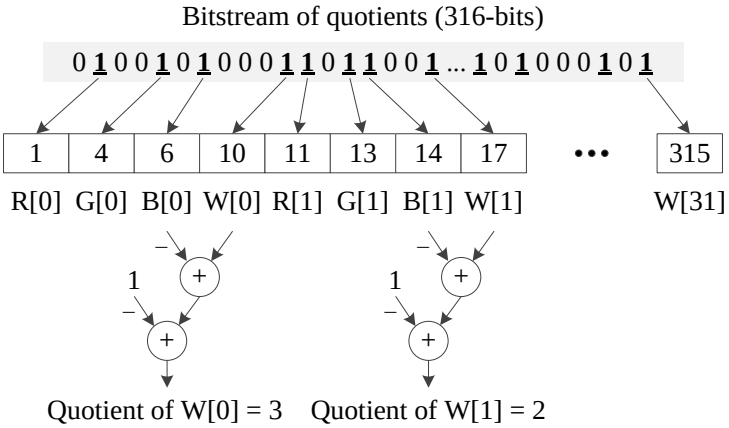


그림 5.8 몫 파싱의 한 예

비트에는 총 128 개의 몫이 존재한다. 몫을 계산하기 위해서는 단일 코드의 경계 비트(1)의 위치를 알 수 있어야 한다. 여러 개의 몫을 파싱하는 방법으로는 크게 두 가지가 있다. 첫 번째는 픽셀 수를 기준으로 몫을 파싱하는 방법이고, 두 번째는 비트 수를 기준으로 몫을 파싱하는 방법이다. 첫 번째 방법의 경우 실제 몫 하나의 최대값이 188(=316-128)이기 때문에 한 사이클에 확인하여야 하는 비트 수가 최대 189이다. 따라서 실시간 하드웨어 구조에 적합하지 않다. 따라서 본 연구는 픽셀 수 기준이 아닌 비트 수 기준으로 파싱을 수행하도록 한다.

그림 5.9는 제안하는 파서 모듈의 블록 다이어그램을 보인다. 제안하는 파서 모듈은 크게 현재 비트스트림에 존재하는 몫을 파싱하는 '파서0'와 결합, 시프팅 등의 작업을 수행하는 '파서1'로 구성된다. 그림 5.7에서 보인 바와 같이 몫에 해당하는 비트는 8 사이클의 첫 번째 사이클에 36 비트, 나머지 사이클에 40 비트씩 들어온다. 이 데이터를 4 비트씩 나눠 각각의 4 비트에 몫이 몇 개 존재하는지, 그 크기가 얼마인지, 다음 4 비

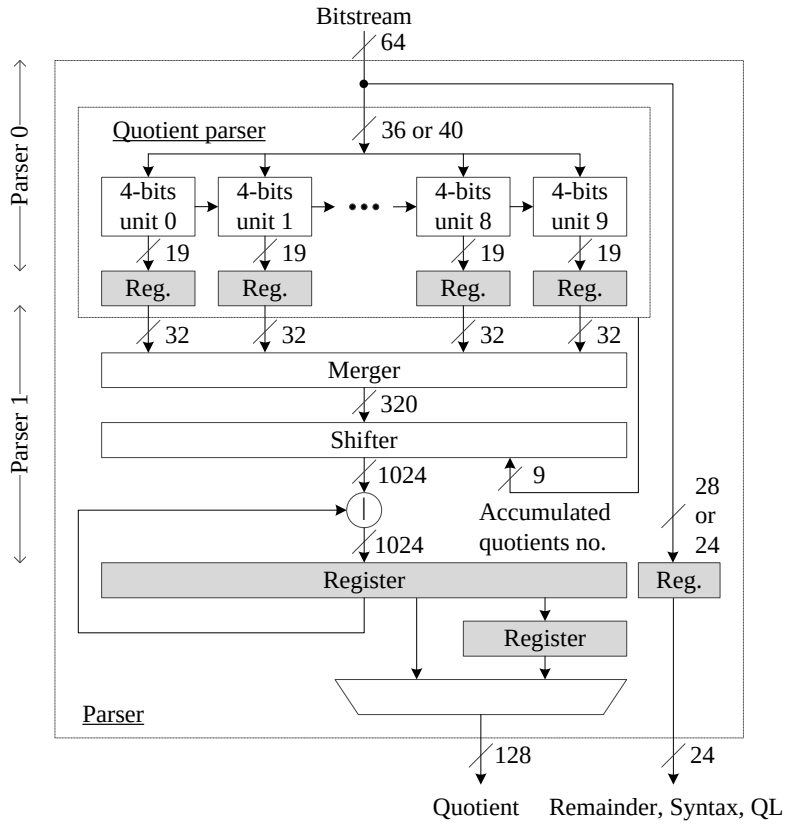


그림 5.9 파서 모듈의 구조

트 유닛에 전달되는 0이 몇 개인지를 확인하도록 한다. 최대 40 비트가 들어오기 때문에 4 비트 유닛은 총 10개가 존재하고, 각 4 비트 유닛에서는 최대 4 개의 몫이 존재할 수 있다. 가능한 몫의 최대 크기가 188(=10111100_2)이므로 하나의 몫은 8 비트로 표현된다. 각각의 4 비트 유닛에서 확인된 몫은 결합된 후 이전까지 누적된 몫의 수만큼 시프팅되어 128 개의 몫을 포함하는 레지스터에 저장된다. 그런 다음 미리 결정된 스케줄에 따라 다음 모듈로 이동한다.

파싱된 몫, QL, 나머지, 2차 차분 신택스 등의 정보는 Golomb-Rice 디

코더에 전달되어 2차 잔차로 복원된다. 복원된 2차 잔차는 1차 잔차 복원 모듈에 전달되어 1차 잔차로 복원된다. 1차 잔차 복원 모듈은 식 (5.5)과 식 (5.6)에 해당하는 연산을 수행한다. 복원된 1차 잔차는 픽셀 복원 모듈에 전달되고, 좌상, 상, 좌측 픽셀값들을 이용하여 최종 복원 픽셀값으로 계산된다. 이때, 윗 라인의 픽셀들을 이용하므로 라인 버퍼로부터 이전 라인 때 저장한 위쪽 4 픽셀을 읽어오도록 한다. 위쪽 4 픽셀 중 가장 오른쪽 픽셀은 다음 4 픽셀 복원을 위하여 따로 레지스터에 저장하도록 한다. 디코더에 도착한 위쪽 픽셀은 QL에 따라 다운-스케일링된다. 매 사이클마다 복원되는 4 픽셀은 최종 RGBW 데이터로 출력된다. 출력되는 데이터는 다음 라인의 디코딩 과정을 위하여 다시 라인 버퍼에 저장된다. 라인 버퍼의 경우 읽고, 쓰기가 동시에 수행되기 때문에 이중-포트 SRAM으로 구성된다.

제안하는 디코더 하드웨어의 타이밍 다이어그램은 그림 5.10과 같다. 그림 5.5와 마찬가지로 가로축은 시간을 의미하고, 점선 한 칸은 1 사이클을 의미한다. 그리고 서로 다른 색의 사각형은 서로 다른 1×32 코딩 블록을 의미한다. 파싱의 경우 현재 비트스트림 파싱에 대한 파싱0 단계와 통합, 시프팅 등으로 구성된 파싱1 단계로 구성된다. 여섯 번의 64 비트 크기의 비트스트림이 파싱되고 나면, 몫에 할당된 비트는 총 80 비트가 남는다. 80 비트가 최대 80 개의 몫을 포함할 수 있으므로 이전 6 사이클 동안 최소 48 개의 몫이 파싱된 것으로 생각할 수 있다. 참고로 5 사이클까지는 최소 8 개(=2 픽셀의 몫 수)의 몫이 파싱되므로 한 사이클에 4 픽셀을 복원하는 것이 불가능하다. 따라서 6 사이클 이후부터 Golomb-Rice 디코더가 1 사이클마다 4 픽셀을 복원하는 동작을 수행하게

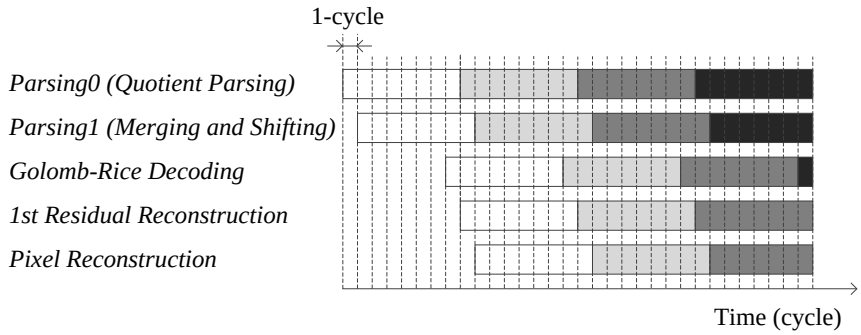


그림 5.10 제안 디코더의 타이밍 다이어그램

된다. 그 뒤에 이어지는 1차 잔차 복원, 픽셀값 복원 등의 모듈들은 연산 결과를 바로 다음 모듈에 전달하는 구조를 갖고, 최종 복원된 픽셀은 1 사이클마다 4 픽셀씩 연속적으로 출력된다.

5.3 실험 결과

이 장은 제안한 알고리즘과 이전 알고리즘의 성능을 비교하도록 한다. 또한 제안 알고리즘에 대한 하드웨어 구현 결과를 보이도록 한다.

5.3.1 알고리즘 실험 결과

모든 실험은 목표 압축률을 2.5로 고정한 후 진행하였고, 이전 방식과 제안 방식의 코딩 블록 크기는 모두 1×32 이다. 실험 이미지의 경우 그림 5.11에서 보인 10 개의 HEVC 실험 시퀀스의 첫 번째 프레임을 10 비트-너비 RGBW 데이터로 변환하여 압축하였다. 본 연구에서는 패널 종

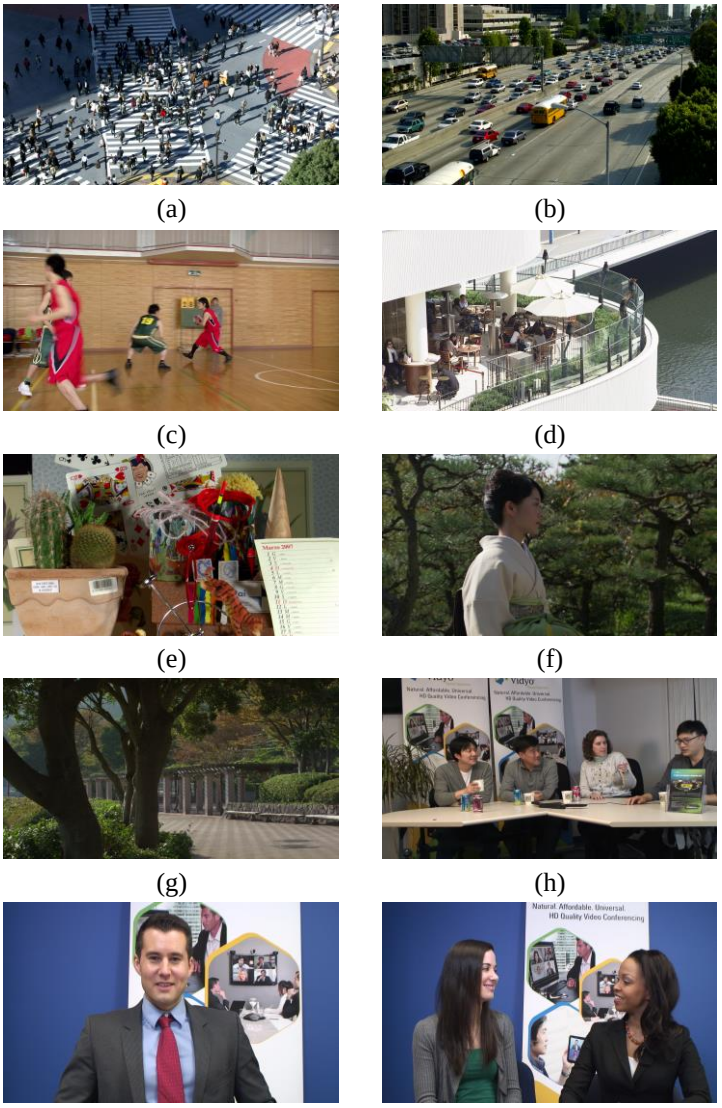


그림 5.11 실험 이미지 (a) People on street(4K) (b) Traffic(4K) (c) Basketball drive(1080p) (d) BQ terrace(1080p) (e) Cactus(1080p) (f) Kimono(1080p) (g) Park scene (1080p) (h) Four people (720p) (i) Johnny(720p) (j) Kristen and Sara(720p)

류에 따라 달라지는 인터-컬러 연관성을 가정하기 위하여 두 종류의 RGB to RGBW 변환식을 실험에 이용하였다. 첫 번째 변환식은 식 (5.8)과 같다[16]. 식 (5.8)에서 R_w, G_w, B_w, W_w는 각각 RGBW 도메인에서의 R, G, B, W 값을 의미한다.

$$\begin{aligned} R_w &= R \\ G_w &= G \\ B_w &= B \\ W_w &= \min(R, G, B) \end{aligned} \tag{5.8}$$

두 번째 변환식은 식 (5.9)와 같다[43-46]. 식 (5.9)에서 r_0, r_1, g_0, g_1, b_0, b_1, w_0 은 상수값이며, 본 연구에서는 이 값들을 1.0, 0.9, 1.0, 0.8, 1.0, 0.5, 0.8으로 설정하였다.

$$\begin{aligned} R_w &= r_0 R - r_1 min(R, G, B) \\ G_w &= g_0 G - g_1 \min(R, G, B) \\ B_w &= b_0 B - b_1 \min(R, G, B) \\ W_w &= w_0 \min(R, G, B) \end{aligned} \tag{5.9}$$

식 (5.8)로 생성된 RGBW 도메인은 RGB 도메인의 특성이 거의 그대로 전달되므로 강한 인터-컬러 연관성을 갖는다. 반면에, 식 (5.9)로 생성된 RGBW 도메인은 상대적으로 낮은 인터-컬러 연관성을 갖는다.

그림 5.12는 이전 예측 방식과 5.1.1장에서 보인 제안 예측 방식의 성능 비교 결과를 보인다. 그림 5.12의 가로축은 식 (5.8)과 식 (5.9)를 이용해 생성한 ‘RGBW 도메인1’과 ‘RGBW 도메인2’를 의미한다. 그리고 세로축은 각각의 예측기에서 계산된 픽셀당 발생하는 평균 절대 잔차의 크기를 의미한다. 이 값의 경우 R, G, B, W 컬러 성분의 평균 절대 잔차를 따로 계산한 다음 그 평균 수치를 표기하였다. DPCM의 경우 좌측의 픽셀만으로 간단히 예측하는 방식이므로 가장 큰 잔차를 보인다. RGBW 도메인1에서는 평균 21.47의 잔차를 보이고, RGBW 도메인2에서는 평균 16.57

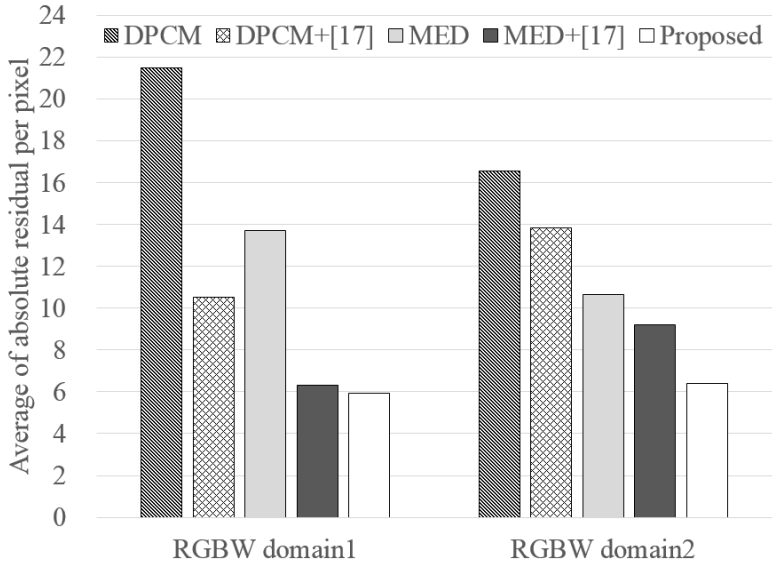


그림 5.12 이전 예측 방식과 제안 예측 방식의 잔차 비교

의 잔차를 보인다. MED의 경우 식 (2.5)에서 보인 JPEG-LS의 고정 예측기를 이용하는 방식이다. 이 경우 이용할 수 있는 공간적 연관성이 확대되면서 두 RGBW 도메인에서의 잔차가 각각 13.71, 10.66까지 작아지게 된다. 기존 방식들의 경우 공간적 연관성만을 이용한다. 반면, 제안 방식은 공간적 연관성과 인터-컬러 연관성 모두를 이용한다. 따라서 조금 더 공정한 비교를 위해 공간적 연관성을 이용하는 DPCM, MED에 인터-컬러 연관성을 이용하는 [17]의 방법을 접목시킨 결과를 함께 보이도록 한다. [17]은 네추럴 이미지에서 각 컬러 성분들의 값이 근사 선형 관계를 갖는다는 점을 이용하여 예측 과정을 수행한다. 가령, R 성분의 참조값을 G 성분의 1차 함수값으로 표현하고, 실제 R 성분의 값에서 이 참조값을 차분하여 잔차를 계산한다. DPCM에 [17]의 방법을 적용할 경우 두 RGBW 도메인에서의 평균 잔차가 각각 10.51, 13.84까지 낮아지게 된다. 이 수치는 [17]을 적용하기 전 결과의 48.95%, 83.52%에 해당하는 수치이다. 그리

고 MED에 [17]의 방법을 적용할 경우 두 RGBW 도메인에서의 평균 잔차가 각각 6.33, 9.18까지 낮아지게 된다. 각각 [17]을 적용하기 전 결과의 46.17%, 86.12%에 해당하는 값이다. 그림 5.12의 흰색 막대 그래프는 제안하는 방식의 결과를 보인다. 제안 방식은 두 RGBW 도메인에서 각각 5.92, 6.38의 평균 잔차를 보인다. 인터-컬러 연관성이 강한 RGBW 도메인 1의 경우 MED+[17]과 제안 방식의 평균 잔차가 각각 6.33과 5.92로 제안 방식이 0.41만큼 작은 평균 절대 잔차를 보인다. 그러나 인터-컬러 연관성이 약한 RGBW 도메인 2의 경우 특정 컬러 성분으로 다른 컬러 성분을 예측하는 [17]의 성능이 감소하므로 기존 방식들의 인터-컬러 연관성 이용 효과가 크게 낮아지게 된다. 반면, 제안 방식의 경우 각 컬러 성분의 근사 평균값을 차분하는 형태로 인터-컬러 연관성을 이용하므로 상대적으로 더 높은 잔차 감소 효과를 보인다. 그 결과, MED+[17]과 제안 방식의 평균 절대 잔차 차이는 2.8로 증가한다. 이와 같이 크기가 감소한 잔차는 인코더의 입력 데이터 크기가 감소되었음을 의미하고, 생성될 비트스트림의 길이 역시 감소될 확률이 높음을 의미한다.

그림 5.13은 이전 방식과 제안 방식의 압축 성능 비교 결과를 보인다. 공정한 비교를 위해 이전 방식과 제안 방식에 동일한 피제수 k를 이용하는 Golomb-Rice 코더를 적용하였다. 단, 이전 방식들의 경우 목표 압축률을 맞출 때까지 QL을 올려가며 반복적으로 코딩을 수행하는 [38]의 반복적 양자화 방법을 이용하였다. 따라서 이전 방식들에 대한 실험 결과는 모두 소프트웨어 시뮬레이션에 대한 결과이다. 본 실험에서는 입력 데이터가 10 비트-너비 RGBW 데이터이므로 화질을 측정하기 위해 식 (5.10)과 같은 PSNR 식을 이용하였다.

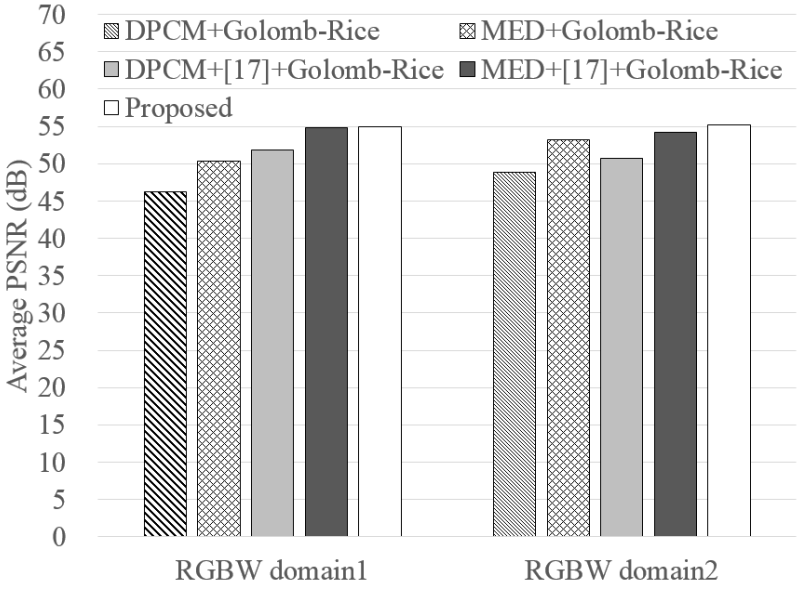


그림 5.13 이전 방식과 제안 방식의 압축 성능 비교

$$PSNR = 10 \times log_{10} \frac{1023^2 \times 4}{MSE_R + MSE_G + MSE_B + MSE_W} \tag{5.10}$$

식 (5.10)에서 MSE_R, MSE_G, MSE_B, MSE_W는 각각 R, G, B, W 컬러 성분에서의 평균 제곱근 오차를 의미한다.

그림 5.13에서 확인할 수 있듯이 제안 방식은 RGBW 도메인1에서 DPCM, MED 기반의 압축 방식들에 비해 각각 8.80 dB, 4.65 dB 향상된 평균 PSNR 결과를 보인다. 그리고 인터-컬러 연관성이 낮은 RGBW 도메인2에서 DPCM, MED 기반의 압축 방식들에 비해 각각 6.42 dB, 2.02 dB 향상된 평균 PSNR 결과를 보인다. 그림 5.13의 방식들이 모두 Golomb 기반의 코더를 이용하므로 이와 같은 성능 향상은 그림 5.12에서 확인한 잔차 감소 효과 때문으로 해석할 수 있다. 공정한 비교를 위해 DPCM, MED에 인터-컬러 연관성을 이용하는 [17]의 방법을 접목시킨 경우 제안 방식이

RGBW 도메인1에서 각각 3.12 dB, 0.11 dB 높은 평균 PSNR 결과를 보인다. 그리고 인터-컬러 연관성이 낮은 RGBW 도메인2에서는 각각 4.54 dB, 1.11 dB 높은 평균 PSNR 결과를 보인다. 실험 결과를 통해 제안 방식이 인터-컬러 연관성이 낮은 RGBW 도메인에서 비교 방식들에 비해 상대적으로 더 높은 압축 효율을 보인다는 사실을 확인할 수 있다.

5.3.2 하드웨어 구현 결과

이 장에서는 제안한 인코더와 디코더의 하드웨어 구현 결과를 확인하도록 한다. 하드웨어는 3840×2160 이미지를 목표로 구현되었으므로 라인 버퍼 크기 역시 해당 이미지 크기에 맞게 설정되었다. 그리고 합성은 0.13μm 공정 라이브러리를 이용하여 진행되었고, 2-NAND 게이트를 기준으로 게이트 수가 계산되었다. 인코더와 디코더의 최대 합성 주파수는 167 MHz이다. 제안 하드웨어의 경우 매 사이클마다 4 픽셀, 4 컬러 성분, 10 비트-너비의 데이터가 들어오거나 나가므로 처리 속도가 26.7 Gbps이다. 표 5.3은 제안한 하드웨어 구조에 대한 게이트 수 결과를 보인다. 인코더의 전체 게이트 수는 56.1K 이다. 이 중 프리-코더와 포스트-코더가 각각 35.8%, 60.4%인 20.1K, 33.9K를 차지한다. 프리-코더의 경우 QL 결정 모듈이 가장 많은 11.8K 를 차지한다. 이는 해당 모듈이 하나의 QL에 대한 2차 잔차 결과를 이용하여 모든 QL에 대한 코딩 결과를 예측하기 때문이다. 포스트-코더의 경우 Golomb-Rice 인코더 모듈이 가장 많은 게이트 수를 이용한다. Golomb-Rice 인코딩 자체는 복잡한 계산을 포함하지 않지만 여러 픽셀을 동시에 처리하고, 패킹에 비트-너비가 큰 시프터를 이용하기 때문에 많은 게이트 수가 이용된다. 디코더 하드웨어의 전체 게

표 5.3 제안 하드웨어의 게이트 수

Module			Gate counts (K gates)	Gate counts (%)
Encoder	Total		56.1	100.0
	Context generator		2.1	3.7
	Pre-coder	Total	20.1	35.8
		Residual generator	8.3	14.8
		QL decision	11.8	21.0
	Post-coder	Total	33.9	60.4
		QL update	5.3	9.4
		Residual generator	8.3	14.8
		Golomb-Rice encoder	15.5	27.6
		Frame memory transmitter	4.8	8.6
Decoder	Total		39.6	100.0
	Address generator		0.2	0.5
	Parser		28.6	72.2
	Golomb-Rice decoder		2.1	5.3
	First residual reconstructor		3.6	9.1
	Pixel residual reconstructor		5.1	12.9

이트 수는 39.6K로 인코더 하드웨어의 전체 게이트 수보다는 그 크기가 작다. 디코더에서 가장 큰 비율을 차지하는 모듈은 파서 모듈이다. 이 모듈은 전달받은 비트스트림을 몫, QL, 나머지, 2차 차분 신택스 정보 등으로 파싱한다. 그런데 그림 5.8에서 언급한대로 몫을 파싱하는데 어려움이 있고, 비트-너비가 큰 레지스터와 시프터가 포함되어 많은 게이트 수가 이용되었다.

표 5.4에서 확인할 수 있듯이 인코더 하드웨어의 내부 메모리로는 라인 버퍼와 딜레이 버퍼가 이용된다. 각각의 버퍼는 이중-포트 SRAM과 단일-포트 SRAM으로 구현되었고, 크기는 각각 153,600 비트와 7,680 비트이다. 디코더 하드웨어의 내부 메모리로는 라인 버퍼가 이용된다. 디코더의 라인 버퍼는 인코더의 라인 버퍼와 메모리의 종류, 크기 등에서 모

표 5.4 제안 하드웨어의 메모리 크기

Module		Memory type	Memory size	
			(bits)	(%)
Encoder	Total	-	161,280	100.0
	Line buffer	Dual port SRAM	(160×960)×1	95.2
	Delay buffer	Single port SRAM	(320×8)×3	4.8
Decoder	Total	-	153,600	100.0
	Line buffer	Dual port SRAM	(160×960)×1	100.0

두 동일하다.

표 5.5는 이전 방식과 제안 방식의 하드웨어 비교 결과를 보인다. 이때, 라인 버퍼 메모리는 모두 제외되었다. [40]은 JPEG-LS 인코더를 하드웨어로 구현한 결과이다. 예측에는 식 (2.5)의 MED 예측기를 이용하고, Golomb 코더를 이용하여 압축을 진행한다. [40]의 경우 성능에 비해 하드웨어 크기를 많이 차지하던 손실 압축 회로를 제외하였다. 따라서 무손실 압축만 가능하다. [34]는 HACP 방식을 이용하여 예측 과정을 수행하고, SBT 방식을 이용하여 코딩을 수행한다. 이때, 코딩 블록의 크기는 8×16 이다. 이 방식의 경우 H.264 압축 표준의 복원 이미지를 재압축하며, 무손실로만 압축이 가능하다. [34]는 하나의 회로를 인코더와 디코더가 공유하여 이용한다. 따라서, 인코더와 디코더가 동시에 동작할 수 없다. 표 5.5에서는 공정한 비교를 위하여 동일한 두 개의 회로가 존재하고, 인코더와 디코더가 하나의 회로를 각각 이용하는 것으로 가정한다. [36]은 AGPM 방식을 이용하여 현재 픽셀과 주변 픽셀간의 관계에 따른 다른 예측 방식을 이용한다. 그리고 각 상황에 따라 Golomb-Rice 코더 또는 이진 코더를 선택적으로 이용한다. [36]의 경우 H.264 압축 표준의 복원 이미지를 재압축하며, 레이트-조절 회로를 포함하고 있으므로 손실 압축도

가능하다. 그러나 손실 압축의 경우 목표 압축률을 정확히 맞추지 못하며, 단순한 양자화 과정에 의해 고압축률에서는 큰 폭의 화질 열화 현상이 발생한다. 그리고 디코더 하드웨어는 처리 속도를 향상시키기 위해 비순차적 디코딩을 수행한다. 따라서 일반적인 디스플레이 장치에서는 해당 방식을 이용할 수 없다. [37]은 MDA 방식을 이용하여 잔차를 생성한다. 그리고 SFL 방식을 이용하여 비트스트림을 생성한다. 이때, 코딩 블록의 크기는 8×8이다. 이 방식의 경우 HEVC 압축 표준의 복원 이미지를 무손실로 재압축한다. [34]와 [37]은 압축 대상, 압축 방법 등이 유사하다. 이 방식들은 압축 표준의 복원 이미지를 재압축하기 때문에 P-프레임에서 발생하는 에러 누적을 피하기 위해 무손실 압축만을 고려한다. 그리고 고속 디코딩 동작을 위해 압축 효율을 낮추는 대신 비트스트림의 길이를 미리 예측할 수 있는 방향으로 압축 알고리즘을 제안하였다. 제안 하드웨어는 RGBW 데이터를 압축하며, RGBW의 인터-컬러 연관성을 이용하는 예측 방식을 이용한다. 그리고 Golomb-Rice 코더를 통해 비트스트림을 생성한다. 이때, 코딩 블록당 목표 압축률을 정확히 맞추기 위해 프리-코더, 포스트-코더로 구성된 두 단계의 인코딩 과정을 수행한다.

제안 방식의 경우 정규화된 처리 속도 대비 이용한 게이트의 수가 인코더는 3.21 Kgates/byte, 디코더는 1.98 Kgates/byte이다. 기존 방식 중 가장 낮은 수치를 보이는 [37]과 비교했을 때, 제안 방식의 인코더는 1.02 Kgates/byte만큼 더 낮고, 디코더는 0.36 Kgates/byte만큼 더 높다. 하지만 제안 방식은 무손실 압축뿐만 아니라 손실 압축도 지원한다. 래스터 주사 순서로 데이터가 들어오는 경우 [37]은 ‘이미지 너비 × 8’에 해당하는 라인 버퍼를 필요로 한다. 반면에 제안 방식의 경우 1-라인 버퍼만을 필요

로 한다. 만약 제안 방식과 같이 1-라인 버퍼만을 이용할 수 있는 상황이라면 세로 방향의 공간적 연관성을 많이 이용하는 [37]의 예측 성능이 크게 감소하게 된다. 또한 [37]의 경우 손실 압축을 하면 비트스트림 내부에서 헤더 비트가 차지하는 비율이 증가하여 압축 효율이 크게 떨어진다. 따라서 표 5.1과 같은 시스템에서는 제안 압축 방식이 더 큰 강점을 갖는다.

표 5.5 이전 방식과 제안 방식의 하드웨어 결과 비교

	Papadonikolakis et al. [40] 1)	Kim et al. [34]		Tsai et al. [36] 3)		Guo et al. [37]		Proposed	
		Encoder	Decoder	Encoder	Decoder	Encoder	Decoder	Encoder	Decoder
Prediction method	MED	HACP		AGPM		MDA		RGBW-ICP	
Coding method	Golomb coding	SBT		GR + binary coding		SFL		GR coding	
Target display resolution	QHD (2560×1440) @30fps	4K (3840×2160) @60fps		4K (3840×2160) @30fps		4K (3840×2160) @60fps		4K (3840×2160) @60fps	
Coding type	Lossless	Lossless		Lossy / Lossless		Lossless		Lossy / Lossless	
Technology (μm)	0.18	0.18		0.18		0.09		0.13	
Frequency (MHz)	183	180		200		300		167	
Throughputs (MB/sec)	183	918	2,556	400		3,200	6,400	3,340	
Normalized throughput (MB/(sec × MHz)	1.00	5.10	14.20	2.00	2.00	10.67	21.33	20.00	20.00
Gate counts (Kgates)	27.7	36.1 2)		20.9	24.4	45.1	34.5	64.2	39.6
Gate counts / normalized throughput (Kgates × (sec × MHz) / MB)	27.7	7.08	2.54	10.45	12.2	4.23	1.62	3.21	1.98
Memory (kB)	2.0	0		0		N/A		1.0	0

1) encoder only 2) encoder and decoder share the single logic and buffer 3) mixed decoding order

제 6 장 압축 성능 및 하드웨어 크기 비교 분석

6.1 압축 성능 비교

이 장에서는 제안 방식 및 비교 방식의 압축 성능을 다양한 환경에서 비교해 보도록 한다. 실험에는 4장에서 제안한 방식이 적용된 1D SPIHT과 2D SPIHT, 가로 방향의 공간적 연관성만을 이용하는 DPCM(only h)-VLC, 가로와 세로 방향의 공간적 연관성을 이용하는 DPCM(v, h)-VLC[38-39], JPEG-LS의 고정 예측기가 이용된 MED-VLC[8], 반고정 길이 압축 방식인 MDA-SFL[37], 5장에서 제안한 ICP-VLC 등이 이용되었다. VLC로는 Golomb-Rice 코딩 방식을 이용하였으며, 공정한 비교를 위해 잘린 비트에 대한 반올림 연산은 모두 제외하였다. 그리고 DPCM-VLC, MED-VLC, MDA-SFL 등은 손실 압축을 지원하기 위해 반복적 양자화 과정을 추가로 적용하였다. 다시 말해, 양자화 파라미터 QL을 1씩 올려가며 목표 압축률을 맞추는 방법을 적용하였다. 1D SPIHT의 경우 코딩 블록의 크기가 1×64이며, 2D SPIHT과 MDA-SFL의 경우 코딩 블록의 크기가 8×8이다. 나머지 압축 방식들의 경우 코딩은 픽셀 단위로 진행하지만 동일한 QL을 갖는 양자화 단위는 1×64로 설정하였다. 실험에는 24개의 Kodak 컬러 이미지를 이용하였으며, 각 결과에는 24개의 평균 수치를 표기하였다. 참고로 Golomb-Rice 코더의 경우 최대 목표 압축률이 제한되어 있다. 가령, 제수가 1인 경우 한 (서브)픽셀당 최소 2비트가 출력된다. 따

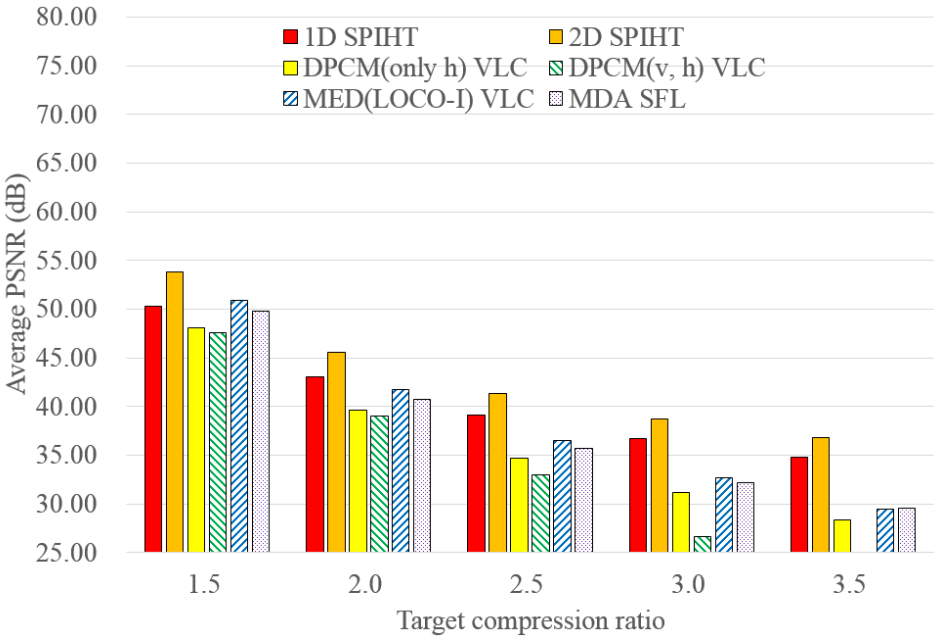


그림 6.1 흑백 이미지에서의 압축 성능 비교

라서 가능한 최대 압축률은 4.0이다. 본 실험에서는 Golomb-Rice 코더의 제수값을 순응적으로 업데이트하는 방식을 이용하였다. 따라서 4.0보다 높은 압축률이 가능하지만 최좌측 픽셀, 헤더 비트 등으로 인해 목표 압축률 4.0이 달성되지 못하는 경우도 발생하였다. 따라서 대다수의 실험에서는 목표 압축률 3.5까지의 결과를 보이도록 한다.

그림 6.1은 흑백 이미지에서의 압축 성능 비교 결과를 보인다. 흑백 이미지를 만들기 위해 24개의 Kodak 컬러 이미지에 식 (3.4)를 적용하였고, 생성된 Y 성분을 입력 이미지로 이용하였다. 그래프의 가로축은 목표 압축률을 의미하고, 세로축은 평균 PSNR을 의미한다. 실험 결과를 통해 목표 압축률이 1.5인 경우 변환(transform) 기반 압축 방식과 예측(prediction) 기반 압축 방식의 성능 차이가 크지 않은 것을 확인할 수 있다. 그러나 압축률이 높아질수록 예측 기반 압축 방식의 성능이 변환 기

반 압축 방식에 비해 상대적으로 크게 떨어지는 것을 확인할 수 있는데, 이는 두 가지 이유로 해석할 수 있다. 첫 번째는 반복적 양자화 과정을 이용하는 예측 기반 압축 방식의 레이트-조절 방식이 목표 압축률을 정확하게 맞추지 못하기 때문이다. 일반적으로 목표 압축률이 높아질수록 선택되는 QL이 증가한다. 그리고 QL이 증가할수록 생성되는 비트스트림 길이의 변화 폭이 커지게 된다. 따라서 목표 압축률이 높아질수록 많은 비트가 버려질 확률이 높고, 그 결과 압축 효율이 크게 감소하게 된다. 두 번째는 변환 기반 압축 방식의 특성 때문이다. 변환 기반 압축 방식은 변환 과정을 통해 이미지의 에너지를 한 쪽으로 모은 후 압축을 수행한다. 따라서 압축에서의 손실이 발생하는 경우 상대적으로 중요하지 않은 부분이 먼저 버려지게 된다. 목표 압축률이 낮은 경우에는 버려지는 비트 자체가 많지 않아 압축 효율 차이가 크지 않다. 그러나 목표 압축률이 높은 경우에는 앞서 설명한 이유로 인해 변환 기반 압축 방식이 예측 기반 압축 방식에 비해 압축 효율 측면에서 더 유리한 특성을 갖는다. 실제 JPEG에서 이와 같은 특성이 이용되고 있다. JPEG의 경우 손실 압축에서는 DCT 기반의 압축 방식을 이용하고, 무손실 압축에서는 예측 기반의 압축 방식을 이용한다[1].

참고로 두 방향의 공간적 연관성을 이용하는 DPCM-VLC의 경우 참조 방향을 알려주기 위한 헤더 비트가 픽셀당 1 비트씩 추가된다. 따라서 가로 방향의 공간적 연관성만을 이용하는 DPCM-VLC에 비해 낮은 압축 효율을 보이게 된다. MDA-SFL의 경우 8×8 블록을 코딩하므로 다른 알고리즘에 비해 이용할 수 있는 공간적 연관성이 크다. 그러나 코딩에서 반고정 길이 압축 방식을 이용하므로 정보가 없는 비트가 많이 포함되어

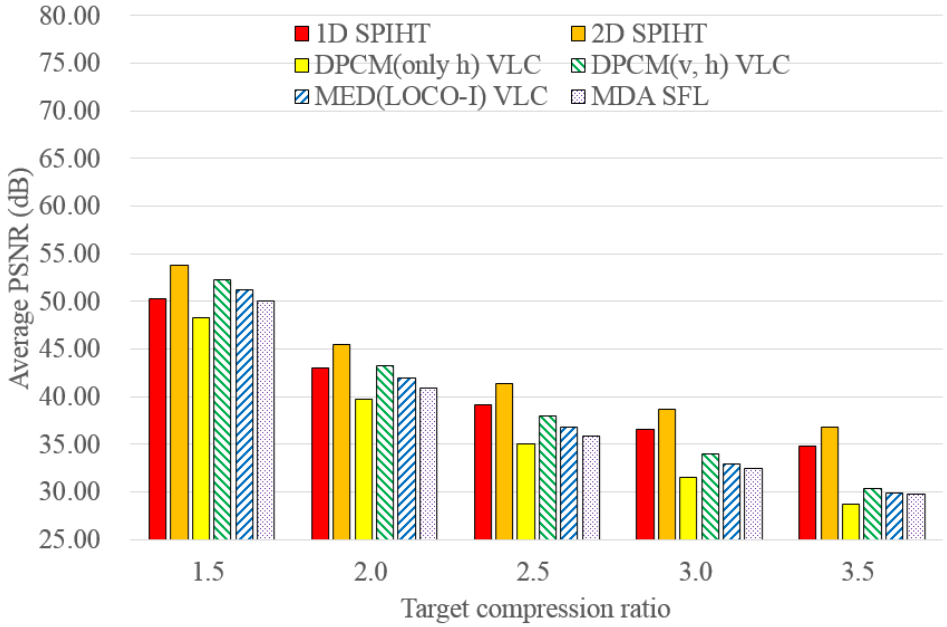


그림 6.2 RGB 444 이미지에서의 압축 성능 비교

낮은 압축 성능을 보이게 되었다.

그림 6.2는 RGB 444 이미지에서의 압축 성능 비교 결과를 보인다. 두 방향의 공간적 연관성을 이용하는 DPCM-VLC의 경우 압축 효율을 향상시키기 위해 세 컬러 성분이 하나의 참조 방향 헤더를 공유하는 형태로 구현하였다. 그 결과, 흑백 이미지 압축에 비해 헤더 비트의 비중이 감소하게 되어 가로 방향의 공간적 연관성만을 이용하는 DPCM-VLC보다 더 높은 성능을 보이게 되었다. 목표 압축률이 2.0 이하인 경우 두 방향의 공간적 연관성을 이용하는 DPCM-VLC가 1D SPIHT보다 더 높은 평균 PSNR 결과를 보인다. 그러나 흑백 이미지에서의 압축 실험과 마찬가지로 목표 압축률이 높아질수록 변환 기반의 압축 방식들이 상대적으로 더 좋은 압축 성능을 보임을 확인할 수 있다.

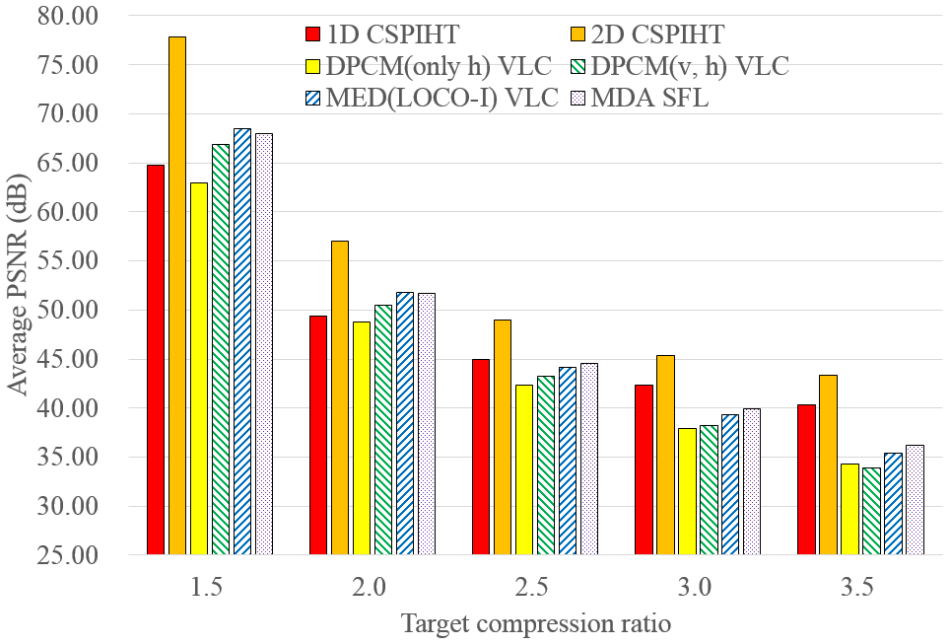


그림 6.3 YCbCr 444 이미지에서의 압축 성능 비교

그림 6.3은 입력 RGB 이미지를 YCbCr 444로 컬러 변환한 후 압축했을 때의 압축 성능 비교 결과를 보인다. 이전 실험과는 달리 인코더 앞단에서 컬러 성분 사이에 존재하는 중복 정보를 제거한다. 따라서 압축 성능이 크게 증가하게 된다. 참고로 SPIHT의 경우 인터-컬러 연관성을 이용하는 Color-SPIHT[47]을 이용하였으므로 'CSPIHT'이라 표기하였다. 이전 실험과 마찬가지로 목표 압축률 2.0 이하에서는 예측 기반 압축 방식이 1D SPIHT보다 더 높은 압축 성능을 보인다. 그러나 목표 압축률 2.5 이후부터는 변환 기반 방식들이 상대적으로 더 높은 압축 효율을 보이는 것을 확인할 수 있다.

그림 6.4는 입력 RGB 이미지를 10 비트-너비 RGBW로 변환한 후 압축했을 때의 압축 성능 비교 결과를 보인다. 이때, 고충실도 상황을 가정

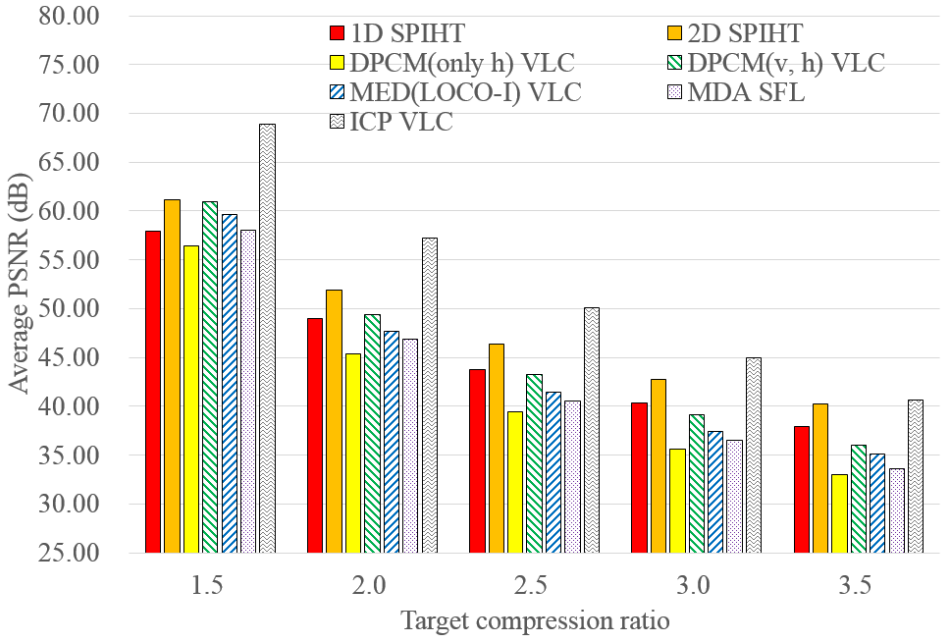


그림 6.4 10 비트-너비 RGBW 이미지에서의 압축 성능 비교

하기 위해 컬러 변환 과정은 배제하였다. 실험 결과를 통해 확인할 수 있듯이 목표 압축률 1.5에서 3.0까지는 ICP-VLC가 다른 압축 방식들에 비해 월등히 좋은 결과를 보인다. 그러나 목표 압축률이 높아질수록 다른 압축 방식들과의 평균 PSNR 간격이 점점 감소하게 되며, 목표 압축률 3.5 이후부터는 변환 기반 압축 방식들이 ICP-VLC보다 더 높은 압축 성능을 보이게 된다. 참고로 8 비트-너비 RGB 이미지에서는 ICP-VLC가 이용하는 인터-컬러 연관성이 더 작기 때문에 목표 압축률 2.5 부근에서부터 변환 기반 압축 방식들이 ICP-VLC보다 높은 압축 성능을 보이게 된다.

그림 6.5는 목표 압축률이 4.0 이상인 경우의 압축 성능 비교 결과를 보인다. 이 장의 초반부에서 설명한 바와 같이 Golomb-Rice 코딩의 경우

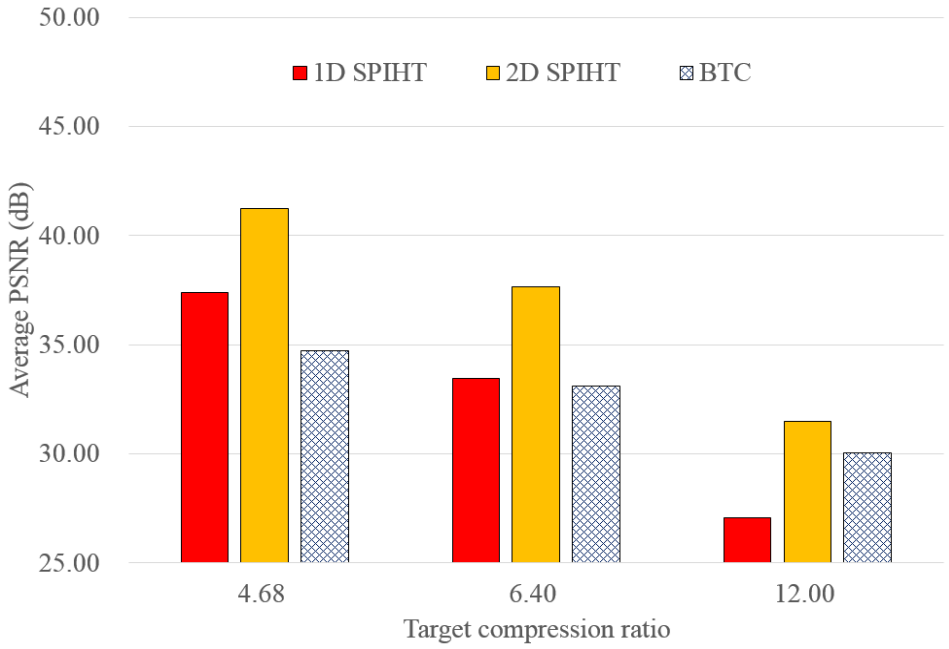


그림 6.5 목표 압축률이 4.0 이상인 경우의 압축 성능 비교

4.0 이상의 목표 압축률을 지원할 수 없다. 따라서 이 실험에서는 예측 기반 압축 방식을 제외하고, 3장에서 제안한 BTC 방식의 결과를 보이도록 한다. 본 실험에서는 표 3.3에서 이용한 4.68, 6.40, 12.00 등을 목표 압축률로 이용하였다. 목표 압축률이 4.68인 경우 SPIHT 방식들이 월등히 좋은 성능을 보인다. 그러나 목표 압축률이 8.0 부근을 넘어가면 분류 비트의 비중이 높아져 SPIHT의 성능이 급격히 떨어진다. 실제로 목표 압축률이 12.00인 경우 BTC 방식이 1D SPIHT보다 2.96 dB 더 높은 평균 PSNR 수치를 보인다.

6.2 하드웨어 크기 비교

이 장에서는 제안 압축 방식 및 비교 압축 방식의 하드웨어 크기를 비교하도록 한다. 그림 6.6은 1D SPIHT, 2D SPIHT, 가로 방향의 공간적 연관성만을 이용하는 DPCM-VLC, 두 방향의 공간적 연관성을 이용하는 DPCM-VLC, MDA-SFL, ICP-VLC, BTC 등의 하드웨어 크기를 비교한 결과를 보인다. 각 압축 방식의 특징은 6.1장의 설명과 동일하다. 그래프의 가로축은 정규화된 처리량 대비 인코더와 디코더의 게이트 수, 라인 메모리 크기 등을 보이고, 세로축은 두 방향의 공간적 연관성을 이용하는 DPCM-VLC의 하드웨어 크기를 1로 했을 때 각 압축 방식의 상대적 하드웨어 크기를 보인다. 라인 메모리의 경우 데이터가 래스터 주사 순서로 전달될 때 요구되는 크기이다. DPCM-VLC, MDA-SFL의 경우 무손실 압축에 대한 하드웨어 크기만을 표기하였다. 따라서 손실 압축까지 고려할 경우 레이트-조절을 위한 회로가 추가되므로 그림 6.6의 수치가 상승한다.

변환 기반 방식인 SPIHT은 상대적으로 많은 정규화된 처리량 대비 게이트 수를 이용한다. 그러나 6.1장에서 확인한 바와 같이 압축 환경, 목표 압축률에 비교적 무관하게 높은 압축 성능을 보인다. 특히, 목표 압축률이 3.0 이상인 경우 예측 기반 압축 방식과의 성능 차이가 크게 벌어지는 것을 6.1장에서 확인하였다. 메모리의 경우 블록 높이에 비례하는 라인 메모리 크기를 필요로 하는 2D SPIHT과는 달리 1D SPIHT은 라인 메모리를 전혀 이용하지 않는다. 따라서 목표 압축률이 3.0 이상이고, 무손실에 가까운 화질을 요구하면서 라인 메모리가 부담스러운 시스템에서는 1D SPIHT이 효과적으로 이용될 수 있다.

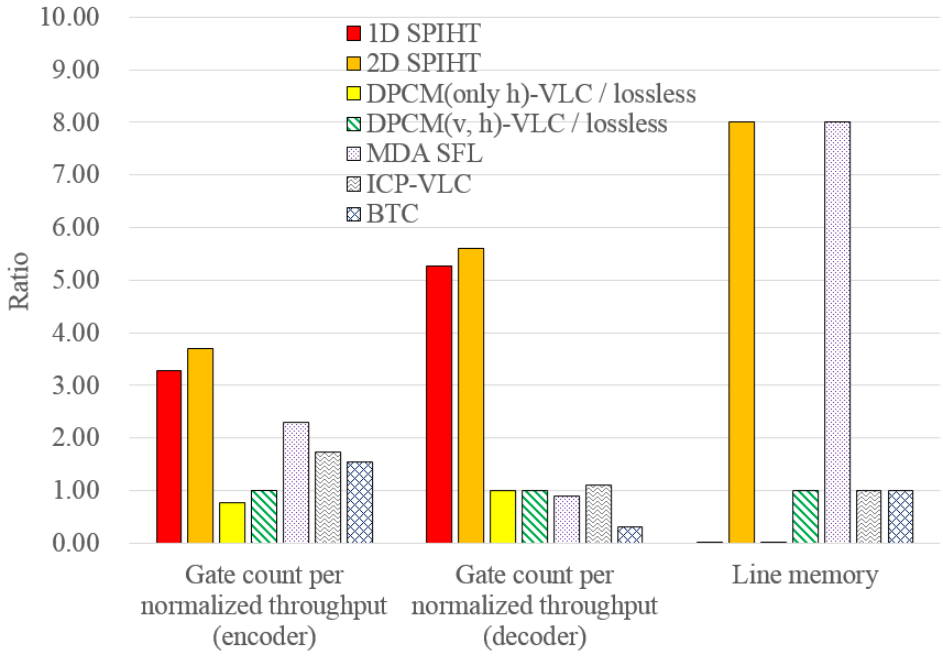


그림 6.6 하드웨어 크기 비교

예측 기반 방식인 ICP-VLC의 경우 손실 압축을 지원하기 위해 2개의 인코더(프리-코더와 포스트-코더)를 이용한다. 따라서 인코더의 정규화된 처리량 대비 게이트 수가 두 방향의 공간적 연관성을 이용하는 DPCM-VLC 방식에 비해 1.74배 더 크다. 하지만 여러 개(예를 들어 8 또는 10)의 QL을 병렬적으로 수행시키고 목표 압축률을 만족시키는 하나의 QL을 선택하는 하드웨어 구조와 비교한다면 그 크기는 매우 작은 편에 속한다. 6.1장에서 확인한 바와 같이 목표 압축률이 2.5 이하인 경우 예측 기반 방식들이 하드웨어 비용 대비 높은 압축 성능을 보인다. 특히, 컬러 변환, 컬러 성분 서브-샘플링이 배제된 고충실도 압축에서 ICP-VLC가 매우 우수한 압축 효율을 보임을 확인하였다. 따라서 목표 압축률이 2.5 이하 수준으로 비교적 낮고, 저가의 하드웨어를 요구하는 시스템에서 ICP-

VLC가 매력적인 옵션이 될 수 있다.

BTC의 경우 기본적으로 매우 단순한 압축 방식이지만 본 논문의 3장에서 제안한 방식의 경우 압축 효율을 높이기 위해 두 종류의 코딩 블록 크기를 이용한다. 따라서 인코더 하드웨어의 크기가 두 방향의 공간적 연관성을 이용하는 DPCM-VLC 방식에 비해 1.55배 더 크다. 그러나 디코더는 헤더에 따라 간단한 파싱만을 수행하므로 0.32배만큼의 하드웨어만을 이용한다. 6.1장에서 확인한 바와 같이 압축률이 12.00인 경우 제안 BTC 방식이 1D SPIHT보다 더 좋은 압축 성능을 보인다. 가장 좋은 성능을 보이는 2D SPIHT은 BTC에 비해 훨씬 큰 하드웨어와 라인 메모리를 요구한다. 따라서 목표 압축률이 높으면서 작고 빠른 압축 하드웨어가 필요한 상황에서는 BTC 방식이 유용하게 이용될 수 있다.

본 논문에서는 비교적 간단한 예측 기반 압축 방식만을 고려하였지만 더 복잡한 예측 기반 압축 방식들이 실제 산업 어플리케이션에서 많이 이용되고 있다. 가령, VESA의 DSC 표준이 한 예이다[35]. 그러나 복잡한 예측 방식은 높은 하드웨어 비용을 요구한다. 표 6.1은 DSC와 제안 1D SPIHT 하드웨어의 FPGA 합성 비교 결과를 보인다. DSC의 경우 최대 4:4:4 포맷, 10 비트-너비를 지원하므로 1-픽셀을 30 비트로 가정하였다. 표 6.1을 통해 확인할 수 있듯이 DSC는 상당히 많은 수의 FPGA 슬라이스를 이용한다. 특히, 압축 효율을 높이기 위해 다양한 기법을 이용하는 인코더에서 상대적으로 많은 하드웨어를 이용한다. 동일 FPGA 슬라이스 수 대비 처리 속도를 비교할 경우 제안 1D SPIHT의 인코더 하드웨어가 DSC의 인코더 하드웨어보다 약 20배 높은 성능을 보인다. 그리고 디코더의 경우 1.42배 높은 성능을 보인다. 또한 제안 1D SPIHT 하드웨어는

표 6.1 DSC 와 제안 1D SPIHT 하드웨어의 FPGA 합성 결과 비교

	VESA DSC 1.1 [48-49]		*Proposed 1D SPIHT (1×64, mono)*	
Architecture	Encoder	Decoder	Encoder	Decoder
Technology	Xilinx Virtex 7-LX330T			
Frequency (MHz)	65	65	106	126
Normalized Throughput (pixels/cycle)	1.00	3.00	4.92	4.57
Throughput (Mbits/sec)	1,950	5,850	5,218	5,760
Slice (no.)	23,425	6,886	3,148	4,774
Throughput / Slice (Mbits/(sec×no.))	0.08	0.85	1.66	1.21
Internal Memory (bit)	0	0	0	0
DSP48	2	1	0	0

DSC가 필요로 하는 라인 버퍼를 이용하지 않는다. 참고로 24개의 Kodak 컬러 이미지를 목표 압축률 3.0으로 압축할 경우 1D SPIHT의 평균 PSNR(36.78 dB)과 DSC의 평균 PSNR(38.84 dB)의 차이가 2.06 dB로 그리 크지 않다. 따라서 가격 대비 높은 압축 효율을 목표로 하는 디스플레이 시스템에서는 1D SPIHT을 포함한 제안 방식들이 매우 큰 강점을 갖는다.

그림 6.7은 6장에서 보인 실험 결과와 분석을 토대로 각 상황에 유리한 압축 방식을 결정한 결과를 보인다. 표의 가로축은 목표 압축률을 의미하고, 세로축은 회로와 메모리 크기를 포함한 하드웨어 크기를 의미한다. 우선 가용 하드웨어가 충분하지 않을 경우 회로의 크기가 매우 작고, 라인 메모리를 이용하지 않는 DPCM(only h)-VLC 방식이 적합하다. 그러나 목표 압축률이 4.0을 넘어갈 경우 목표 압축률을 달성하지 못하는 경우가 발생한다. 이 경우 1D SPIHT 또는 BTC 방식을 이용할 수 있다. 1D SPIHT은 압축 효율 측면에서 유리하고, BTC는 하드웨어 크기 측면에서 유리하다. 그러나 목표 압축률이 8.0을 넘어갈 경우 1D SPIHT의 압축 성능이 급격히 떨어지므로 BTC 방식을 이용하는 것이 유리하다. 이용할 수 있는 메모리가

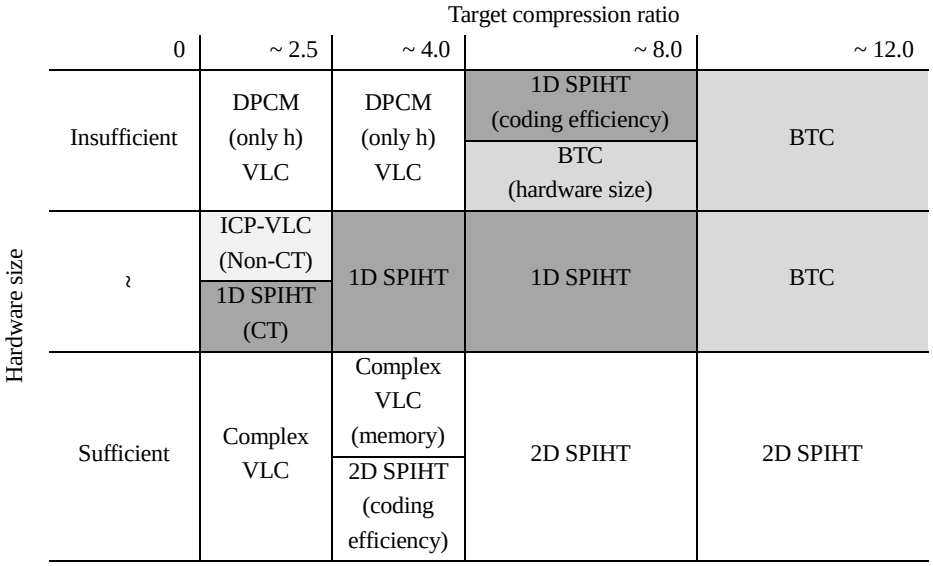


그림 6.7 상황에 따른 압축 방식 선택

부족할 경우 BTC는 높이가 1인 블록을 코딩할 수도 있다. 가용 하드웨어가 증가할 경우 선택할 수 있는 옵션의 수가 증가한다. 목표 압축률이 2.5 이하이고 컬러 변환(color transform; CT)을 이용하지 못하는 경우 ICP-VLC를 이용하는 것이 하드웨어 비용 대비 압축 효율 측면에서 유리하다. 반면, 목표 압축률이 2.5 이하이고 컬러 변환을 이용할 수 있는 경우 1D SPIHT을 이용하는 것이 더 유리하다. 목표 압축률이 2.5 이상이고 8.0 미만인 경우 1D SPIHT을 이용하는 것이 유리하다. 그러나 목표 압축률이 8.0을 넘어가면 1D SPIHT의 압축 효율이 급격히 감소하므로 BTC 방식을 이용하는 것이 압축에 더 유리하다. 가용 하드웨어가 충분한 경우 VESA DSC와 같은 복잡한 예측 기반 VLC 또는 2D SPIHT을 이용할 수 있다. 2D SPIHT은 목표 압축률이 4.0 이상인 경우 압축 효율 측면에서 유리하다. 반면, 복잡한 예측 기반 VLC 방법은 목표 압축률이 낮거나 라인 메모리를 절약하고 싶을 때 또는 짧은 지연 시간이 필요할 때 유리하게 이용될 수 있다.

제 7 장 결론

본 연구는 디스플레이 장치에서의 여러 가지 압축 알고리즘들을 제안하고, 해당 알고리즘들을 하드웨어로 구현하였다. 디스플레이 장치에서의 압축 방식은 일반적인 비디오 압축 표준과는 다른 특수한 어플리케이션을 목표로 한다. 본 연구에서는 LCD 오버드라이브, 저비용 근접-무손실 프레임 메모리 압축, RGBW 컬러 이미지 압축 등의 어플리케이션을 목표로 하였다. 이때, 각각의 압축 어플리케이션에서는 BTC, 1D SPIHT, 예측 기반의 알고리즘들이 이용되었다. 제안 알고리즘들은 래스터 주사 순서에 적합한 구조이며, 압축 단위당 목표 압축률을 정확히 맞출 수 있는 특징이 있다.

제안 BTC 알고리즘은 1/12 압축을 목표로 하고, 디스플레이 시스템의 래스터 주사 순서에 적합하도록 높이가 2인 코딩 블록을 이용한다. 고압축률과 낮은 공간적 연관성에 의해 낮아진 압축 효율을 향상시키기 위해 2×16 기본 블록을 하나의 2×16 코딩 블록 또는 두 개의 2×8 코딩 블록을 이용하여 코딩한다. 제안 비트-절약 방식은 공간적 연관성을 이용하여 RV를 표현하기 위해 이용되는 비트의 수를 감소시킨다. 절약된 비트는 압축률이 낮은 두 개의 2×8 코딩 블록을 이용하는데 쓰인다. 따라서 목표 압축률을 만족시키면서 압축 효율을 향상시킬 수 있다. 제안 방식은 24 개의 Kodak 컬러 이미지에서 30.03 dB의 평균 PSNR 결과를 보인다. 제안 알고리즘에 대한 하드웨어 구현의 경우 비용 효율이 높은 프레임 메모리 인터페이스 구조가 제안되었다. 제안 인코더, 디코더 하드웨어의 경우 각각 143 MHz와 333 MHz에서 27.5 Gbps와 63.9 Gbps의 처리 속도를

보인다. 인코더 하드웨어의 경우 21,312 비트의 내부 메모리를 이용하며, 디코더 하드웨어의 경우 5,952 비트의 내부 메모리를 이용한다. 그리고 인코더 하드웨어와 디코더 하드웨어는 각각 68.7K 게이트와 13.6K 게이트를 이용한다.

제안 1D SPIHT 구조는 크게 두 가지의 부분에서 새롭게 제안되었다. 첫 번째는 고속 동작을 위한 방식이다. 그리고 두 번째는 고속 동작을 위해 희생된 압축 효율 향상 방식이다. 첫 번째 내용을 위해서는 여러 주파수 밴드 사이에 존재하는 의존 관계를 해결해야 한다. 이를 위하여 해당 의존 관계를 분석하고, 이를 해결할 수 있는 파이프라인 스케쥴 방식을 제안하였다. 또한 디코더에서는 각 패스가 디코딩할 비트스트림의 시작 주소를 미리 계산할 수 있는 방법을 제안하였다. 그 결과, 인코더 하드웨어와 디코더 하드웨어는 각각 7.04 Gbps와 7.63 Gbps의 처리 속도를 보이게 되었다. 그러나 앞서 제안한 방식은 24 개의 Kodak 컬러 이미지에 대하여 평균 1.40 dB의 PSNR 감소를 요구한다. 감소된 압축 성능을 향상시키기 위한 두 번째 내용에서는 더미 비트 재사용 방법, 절반-패스 처리 방법, 분류 비트 재배치 방법 등을 제안하였다. 그 결과, 24 개의 Kodak 컬러 이미지에 대하여 0.96 dB만큼의 평균 PSNR이 향상되었다. 제안 구조는 기본적으로 고속 1D SPIHT을 목표로 한다. 그러나 여러 주파수 밴드에 대한 병렬 처리 등의 방법은 2D SPIHT의 처리 속도를 향상시킬 때에도 동일하게 이용될 수 있다.

RGBW 컬러 이미지 압축을 위한 제안 알고리즘은 늘어난 RGBW 인터-컬러 연관성을 이용하여 예측 과정을 수행한다. 제안하는 예측 과정은 크게 두 단계로 구성된다. 첫 번째 단계에서는 공간적 연관성을 이용하여

잔차를 계산한다. 두 번째 단계에서는 RGBW 인터-컬러 연관성을 이용하여 추가로 잔차를 감소시킨다. 제안 예측 방식은 기존 방식들에 비해 60% 이하의 잔차를 보인다. 제안하는 코딩 방식은 Golomb-Rice 코딩을 기반으로 한 고정 길이 압축 방식이다. 이를 위해 제한된 상황에 대하여 실제 인코딩을 진행한 후 다른 모든 상황에 대한 예측 인코딩 정보를 계산하는 프리-코더, 전달받은 예측 정보를 이용하여 실제 인코딩을 수행하는 포스트-코더로 인코딩 과정을 구성한다. 목표 압축률이 2.5인 경우 제안 알고리즘은 10 개의 HEVC 실험 이미지에 대하여 55.26 dB의 평균 PSNR을 보인다. 제안 알고리즘에 대한 하드웨어 구현의 경우 인코더와 디코더 모두 26.7 Gbps의 처리 속도를 보인다. 이때, 프리-코더와 포스트-코더로 구성된 인코더 하드웨어의 경우 64.2K 게이트를 이용하며, 디코더 하드웨어의 경우 39.6K 게이트를 이용한다.

참고문헌

[1] G. K. Wallace, "The JPEG Still Picture Compression Standard," *IEEE Trans. Consum. Electron*., vol. 38, no. 1, pp. xviii-xxxiv, Feb. 1992.

[2] C. Christopoulos, A. Skodras, and T. Ebrahimi, "The JPEG2000 Still Image Coding System: An Overview," *IEEE Trans. Consum. Electron*., vol. 46, no. 4, pp. 1103-1127, Nov. 2000.

[3] E. J. Delp and O. R. Mitchell, "Image Compression Using Block Truncation Coding," *IEEE Trans. Commun.*, vol. 27, no. 9, pp. 1335-1342, Sep. 1979.

[4] A. Said and W. A. Pearlman, "A New, Fast, and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees," *IEEE Trans. Circuits Syst. Video Technol*., vol. 6, no. 3, pp. 243-250, Jun. 1996.

[5] S. Ktata, K. Ouni, and N. Ellouze, "A Novel Compression Algorithm for Electrocardiogram Signals based on Wavelet Transform and SPIHT," *Int. J. Signal Process.*, vol. 5, no. 4, pp. 253-158, Spr. 2009.

[6] Z.-H. Zhang and J. Zhang, "Unsymmetrical SPIHT Codec and 1D SPIHT Codec," in *Proc. Int. Conf. Electr. Control Eng.*, pp. 2498-2501, Jun. 2010.

[7] S. Kim, D. Lee, H. Kim, N. X. Truong, and J.-S. Kim, "An Enhanced One-Dimensional SPIHT Algorithm and Its Implementation for TV Systems," *Displays*, vol. 40, pp. 68-77, Dec. 2015.

[8] M. J. Weinberger, G. Seroussi, and G. Sapiro, “The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS,” *IEEE Trans. Image Process*., vol. 9, no. 8, pp. 1309-1324, Aug. 2000.

[9] J. Someya, N. Okuda, and H. Sugiura, "The Suppression of Noise on A Dithering Image in LCD Overdrive," *IEEE Trans. Consum. Electron*., vol. 52, no. 4, pp. 1325-1332, Nov. 2006.

[10] S.-T. Wu and C.-S. Wu, “Small Angle Relaxation of Highly Deformed Nematic Liquid Crystals,” *Appl. Phys. Lett*., vol. 53, no. 19, pp. 1794-1796, Sep. 1988.

[11] S.-T. Wu, “Nematic Liquid Crystal Modulator with Response Time Less Than 100 μm at Room Temperature,” *Appl. Phys. Lett*., vol. 57, no. 10, pp. 986-988, Jun. 1990.

[12] H. Sasaki, T. Ishikawa, Y. Ishikawa, K. Ichikawa, and N. Saitou, "Flicker-reduced Memory Compression for a Volume-zone Liquid Crystal Display Overdrive," *J. Inf. Display*, vol. 12, no. 2, pp. 77-83, Jun. 2011.

[13] J. Someya, M. Yamakawa, N. Okuda, T. Kimura, M. Yoshida, and E. Gofuku, “Reduction of Memory Capacity in Feedforward Driving by Image Compression,” in *SID Symp. Dig. Tech. Papers*, vol. 33, no. 1, pp. 72-75, May 2002.

[14] J. Someya, N. Okuda, H. Yoshii, M. Yamakawa, H. Oura, A. Minami, and H. Tachibana, “A New LCD-Controller for Improvement of Response Time by Compression FFD,” in *SID Symp. Dig. Tech. Papers*, vol. 34, no. 1, pp. 1346-1349, May 2003.

[15] C. M. Wu, V. Wang, and C. Doung, "Selective Use of LCD Overdrive for Reducing Motion Artifacts in an LCD Device." U.S. Patent 7,696,988, Apr. 13, 2010.

[16] C. H. B. Elliott, T. L. Credelle, and M. F. Higgins, “Adding a White Subpixel”, *SID Inf. Display*, vol. 21, no. 5, pp. 26-31, 2005.

[17] B. C. Song, Y. G. Lee, and N. H. Kim, “Block Adaptive Inter-Color Compensation Algorithm for RGB 4:4:4 Video Coding”, *IEEE Trans. Circuits Syst. Video Technol.*, vol. 18, no. 10, pp. 1447-1451, Oct., 2008.

[18] Y.-H. Kim, B. Choi, and J. Paik, “High-Fidelity RGB Video Coding Using Adaptive Inter-Plane Weighted Prediction”, *IEEE Trans. Circuits Syst. Video Technol*., vol. 19, no. 7, pp. 1051-1056, Jul. 2009.

[19] C. Lee and V. Monga, “Power-Constrained RGB-to-RGBW Conversion for Emissive Displays,” in *Proc. IEEE Int. Conf. on Acoustic, Speech, and Signal Processing*, pp.1205-1209, May 2014.

[20] Y.-T. Kim, D.-S. Park, and Y. Park, “System and Method for Converting RGB to RGBW Color Using White Value Extraction,” U.S. Patent 8,035,655, Oct. 11, 2011.

[21] Y. Wu and D. C. Coll, "Single Bit-map Block Truncation Coding of Color Images," *IEEE J. Select. Areas Commun.*, vol. 10, no. 5, pp. 952-959, Jun. 1992.

[22] M. D. Lema and O. R. Motchell, “Absolute Moment Block Truncation Coding and Its Application to Color Images”, *IEEE Trans. Commun*., vol. 32, no. 10, pp. 1148-1157, Oct. 1984.

[23] H. B. Kekre and S. D. Thepade, “Image Retrieval Using Augmented Block Truncation Coding Techniques,” in *Proc. Int. Conf. on Advances in Computing, Commun. Control*, pp. 384-390, 2009.

[24] J. A. Hartigan and M. A. Wong, "A K-means Clustering Algorithm," *Appl. Statistics*, pp. 100-108, 1979.

[25] C. Ding and X. He, "K-means Clustering via Principal Component Analysis," in *Proc. 21st Int. Conf. Machine Learning*, pp. 225-232, Jul. 2004.

[26] T. Kurita and N. Otsu, “A Method of Block Truncation Coding for Color Image Compression,” *IEEE Trans. Commun.*, vol. 41, no. 9, pp. 1270-1274, Sep. 1993.

[27] J.-W. Han, M.-C. Hwang, S.-G. Kim, and T.-H. You, “Vector Quantizer based Block Truncation Coding for Color Image Compression in LCD Overdrive,” *IEEE Trans. Consum. Electron.*, vol. 54, no. 4, pp. 1839-1845, Nov. 2008.

[28] J. Wang and J.-W. Chong, "Adaptive Multi-level Block Truncation Coding for Frame Memory Reduction in LCD Overdrive," *IEEE Trans. Consum. Electron.*, vol. 56, no. 2, pp. 1130-1136, May 2010.

[29] F. W. Wheeler and W. A. Pearlman, “SPIHT Image Compression Without Lists,” in *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, vol. 6, pp. 2047-2050, Jun. 2000.

[30] P. Corsonello, S. Perri, G. Staino, M. Lanuzza, and G. Cocorullo, "Low Bit Rate Image Compression Core for Onboard Space Applications," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 16, no. 1, pp. 114-128, Jan. 2006.

[31] C.-C. Cheng, P.-C. Tseng, C.-T. Huang, and L.-G. Chen, "Multi-Mode Embedded Compression Codec Engine for Power-Aware Video Coding System," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 19, no. 2, pp. 141-150, Feb. 2009.

[32] T. W. Fry and S. A. Hauck, "SPIHT Image Compression on FPGAs," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 9, pp. 1138-1147, Sep. 2005.

[33] Y. Jin and H.-J. Lee, "A Block-based Pass-parallel SPIHT Algorithm*," IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 7, pp. 1064-1075, Jul. 2012.

[34] J. Kim and S.-M. Kyung, "A Lossless Embedded Compression Using Significant Bit Truncation for HD Video Coding," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 20, no. 6, pp. 848-860, Jun. 2010.

[35] "VESA Display Stream Compression," [Online]. Available: http://www.vesa.org/wp-content/uploads/2014/04/VESA_DSC-ETP200.pdf (Accessed on 4 Jan. 2016)

[36] T.-H. Tsai and Y.-H. Lee, "A 6.4 Gbit/s Embedded Compression Codec for Memory-Efficient Applications on Advanced-HD Specification," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 20, no.10, pp. 1277-1291, Oct. 2010.

[37] L. Guo, D. Zhou, and S. Goto, "A New Reference Frame Recompression Algorithm and Its VLSI Architecture for UHD TV Video Codec," *IEEE Trans. Multimedia*, vol. 16, no. 8, pp. 2323-2332, Dec. 2014.

[38] Y. Lee, C.-E. Rhee, and H.-J. Lee, “A New Frame Recompression Algorithm Integrated with H.264 Video Compression,” in *Proc. IEEE Int. Symp. Circuits Syst.*, pp. 1621-1624, May 2007.

[39] Y. Jin, Y. Lee, and H.-J. Lee, “A New Frame Memory Compression Algorithm with DPCM and VLC in a 4x4 Block,” *EURASIP J. Adv. Signal Process*., vol. 2009, Article ID 629285, 18 pages, Feb. 2009.

[40] M. Papadonikolakis, V. Pantazis, and A. P. Kakarountas, “Efficient High-Performance ASIC Implementation of JPEG-LS Encoder”, in *Proc. Design, Automation & Test in Eur. Conf. Exhibition,* pp. 1-6, 2007.

[41] “Kodak lossless true color image suite,” [Online]. Available: http://r0k.us/graphics/kodak/ (Accessed on 12 Jan. 2016)

[42] S. M. Saif, S. Nassar, H. M. Abbas, and A. T. Soliman, “FPGA Implementation of Block Truncation Coding Algorithm for Gray Scale and Color Images,” in *Proc. IEEE Can. Conf. Electrical and Comput. Eng.*, vol. 1, pp.23-26, May 2003.

[43] H. Tanioka, "Image Processing Apparatus Which Extracts White Component Data," U.S. Patent 5,929,843, Jul. 27, 1999.

[44] C.-C. Lai and C.-C. Tsai, “A Modified Stripe-RGBW TFT-LCD with Image-Processing Engine for Mobile Phone Displays”, *IEEE Trans. Consum. Electron*., vol. 53, no. 4, pp. 1628-1633, Nov. 2007.

[45] B. Lee, C. Park, S. Kim, T. Kim, Y. Yang, J. Oh, J. Choi, M. Hong, D. Sakong, K. Chung, S. Lee, and C. Kim, “TFT-LCD with RGBW Color System”, in *SID Symp. Dig. Tech. Papers*, vol. 34, no. 1, pp. 1212-1215, May 2003.

[46] L. Wang, Y. Tu, L. Chen, K. Teunissen, and I. Heynderickx, “Trade-off between Luminance and Color in RGBW Displays for Mobile-phone Usage,” in *SID Symp. Dig. Tech. Papers*, vol. 38, no. 1, pp. 1142–1145, May 2007.

[47] A. A. Kassim and W. S. Lee, “Embedded Color Image Coding Using SPIHT with Partially Linked Spatial Orientation Trees,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 2, pp. 203-206, Feb. 2003.

[48] “VESA DSC 1.1 Encoder IP Core,” [Online]. Available: http://www.xilinx.com/products/intellectual-property/1-8effft.html#metrics (Accessed on 4 Jan. 2016)

[49] “VESA DSC 1.1 Decoder IP Core,” [Online]. Available: http://www.xilinx.com/products/intellectual-property/1-5wjzl9.html#metrics (Accessed on 4 Jan. 2016)

Abstract

Compression for a display device has special features which are different from those of general video compression standards. First, it aims for special applications. Second, its hardware size is small and target compression ratio is low, relatively, because of high compression gain, low power consumption, and real-time processing. Third, it should be suitable for the Raster-scan order. Fourth, in order to restrict the frame memory size and apply the random memory access, the length of the coded bitstream should fit into the target bit-length exactly. In this paper, three compression algorithms and hardware designs which satisfy these features are proposed.

For an LCD overdrive application, a BTC(block truncation coding)-based compression method is proposed. In order to reduce the frame memory size further, the target compression ratio is set by 12 that is twice of previous methods. Moreover, two schemes are proposed to improve compression efficiency. First, a bit-saving scheme that utilizes spatial correlation between vertically adjacent blocks is proposed. Second, an adaptive block size scheme in which one 2×16 coding block is used for blocks with low image complexity while two 2×8 coding blocks are used for blocks with high image complexity is proposed. To achieve the target compression ratio, the 2×8 coding blocks utilize bits which are saved by the bit-saving scheme.

For a low-cost and near-lossless frame memory compression application, a 1D SPIHT(set partitioning in hierarchical trees)-based compression method is proposed. The conventional SPIHT is an effective algorithm for a fixed compression ratio. However, its 1D variation has received little attention, even though its 1D nature is

suitable for the Raster-scan order. This paper proposes a hardware design which resolves a slow speed problem of the previous 1D SPIHT. The 1D SPIHT algorithm is modified to exploit parallelism. For the encoder, dependences that prohibit the parallel processing are analyzed and resolved. For the decoder, a scheme of which each pass predicts the number of coded bits prior to decoding is proposed, thus parallel and pipelined executions are available.

For a high-fidelity RGBW color image compression application, a prediction-based compression method is proposed. The proposed method is composed of two prediction steps. The first step utilizes spatial correlation while the second step utilizes inter-color correlation. In order to code residuals, the proposed method uses a VLC(variable length coding) which is generally adopted in video compression standards for high compression efficiency. However, previous VLC methods have difficulty in fitting the length of the coded bitstream into the target bit-length. This paper proposes a fixed ratio compression method which is based on the Golomb-Rice coding. The proposed encoder is composed of two sub-coders, pre-coder and post-coder. The pre-coder codes input data in the certain situation and calculates predicted encoding information for all other situations. By using the information, the suitable quantization level is selected and the post-coder generates the final bitstream.

Keywords : liquid-crystal display(LCD), color image compression, real-time compression, VLSI implementation

Student Number : 2012-30197