석 사 학 위 논 문

시스템 온 칩에서의 부하 균형을 위한 적응적 주소 매핑

김 소 연

제주대학교 대학원

에너지응용시스템학부 전자공학전공

2024년 02월

시스템 온 칩에서의 부하 균형을 위한 적응적 주소 매핑

이 논문을 공학석사 학위논문으로 제출함

김 소 연

제주대학교 대학원

에너지응용시스템학부 전자공학전공

지 도 교 수 허 재 영

김소연의 공학 석사 학위논문을 인준함

2023년 12월

심사위원장 고석주 (인)
위 원 허재영 (인)
위 원 오동렬 (인)

목차

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AXI	Advanced eXtensible Interface
CIAM	Cache Interleaved Address Map
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
DSP	Digital Signa Processor
FF	Flip Flop
FPGA	Field Programmable Gate Array
GSLN	Grouped super-line Number
GSLS	Grouped super-line Size
IO	Input Output
IP	Intellectual Property
LIAM	Linear Address Mapping
LUT	Look Up Table
pLIAM	padded Linear Address Map
SoC	System on Chip
SLN	Super-line Number
SLS	Super-line Size

시스템-온-칩에서의 부하 균형을 위한 적응적인 주소 매핑

김 소 연

제주대학교 대학원 에너지응용시스템학부 전자공학전공

초록

시스템 온 칩에서 이차원 데이터를 처리할 때 기존의 주소 맵은 온칩 메모리 구성 요소에 트래픽 정체를 초래하는 경우가 있다. 또한 애플리케이션의 접근 패턴이 주소 맵과 일치하지 않으면 메모리 활용도가 저하될 수 있다. 이러한 트래픽 정체를 줄이고 메모리 시스템 성능을 개선하기 위해 본 논문에서는 주소 매핑과 하드웨어 구성에 대해 적응적으로 이를 해결하는 방안 두 가지를 제안한다. 첫째, 소프트웨어의 요소만을 변경하여 이미지 크기를 패딩하는 기법을 기술한다. 둘째, 하드웨어의 요소를 변경하여 주소의 캐시 비트를 전환하는 기법을 기술한다. 이와 같은 두 가지의 기법들을 적용할 때 메모리 구조 전반에 부하 균형이 향상될 수 있다. 주로 고대역폭 영상처리 애플리케이션을 대상으로 한 방식을 설계하고 메모리 성능 향상을 확인하는 실험을 수행한다. 또한 이에 따른 오버헤드를 측정한다. 결과적으로 실험에 따르면 제시된 설계는 부하 균형률이 향상되고 오버헤드(메모리 공간 또는 하드웨어 비용)가 크지 않음을 보인다.

제 1 장 서론

1.1 동기

시스템 온 칩 (System on Chip, SoC)은 하나의 칩에 전체의 시스템을 집약적으로 담은 것이다. 현대 시스템 온 칩에는 여러 개의 메모리와 캐시들을 내장하고 있다. 이때 메모리와 프로세서 간의 성능 차이가 발생할 수 있다. 이러한 차이를 줄이기 위해 현대 시스템 온 칩에서는 다중의 DRAM (Dynamic Random Access Memory) 채널을 적용하였다. 또한 행 버퍼 충돌 (Row-buffer conflict)을 줄이고 DRAM 이용률 (Utilization)을 개선하기 위해 인터리빙 기술이 널리 사용되었다[1-8]. 이러한 기술들을 사용한 DRAM 부품들은 트랜잭션(Transaction)들을 병렬적으로 다룰 수 있고 부하를 균형 있게 처리할 수 있고, 시스템 이용률을 개선할 수 있다. 최신의 시스템 온 칩에는 다수의 캐시가 내장된다. 메모리와 프로세서 간의 성능 차이를 줄이기 위해 캐시 성능 개선에 관한 연구들도 진행되어왔다[9-12]. 이러한 연구들은 캐시 성능이 향상되어 데이터 처리 속도를 빠르게 하였다. 그러나 이 캐시와 DRAM 채널을 동시에 고려한 연구는 거의 없었다. 본 논문에서는 부하 균형이 향상되기 위해 메모리의 채널과 캐시를 고려한 매핑 방식을 제안한다.

1.2 범위

임베디드 시스템에서 이미지 처리 애플리케이션을 구동할 때 이미지 픽셀들은 메모리 주소들을 가지고 있다. 주소 맵은 픽셀의 2차원 좌표와 메모리 주소를 매핑하는 데 사용된다. 본 논문에서는 이미지 처리와 같은 2D 데이터 애플리케이션을 대상으로 하였다. 그 이유는 다음과 같다.

- 이미지 처리를 진행할 때 높은 대역폭과 부하를 균형이 있게 사용하는 것은

매우 중요한 문제이다.

- 트래픽 패턴은 규칙적이고 잘 알려져 있다. 시스템에서 일관성 있는 효과를 얻기 위해 잘 알려진 트래픽 패턴을 사용하였다.

기존의 선형 주소 매핑 방식은 선형으로 데이터를 처리할 때 자주 쓰이는 방식인데 이를 적응적으로 주소 매핑하는 방식으로 개선하여 성능 향상을 도모하였다. 또한 이를 위해 소프트웨어 변경 방식과 하드웨어 변경 방식 두 가지를 각각 적용하여 성능 향상을 목표로 하였다.

1.3 기존 방식의 문제점

데이터를 처리할 때 보통 기존의 선형 주소 매핑 (LIAM, Linear Address Mapping) 방식이 주로 사용된다. LIAM 방식으로 데이터를 선형으로 처리할 경우의 예 중 하나인 래스터 스캔 (Raster scan) 방식일 경우 데이터를 처리할 때 매핑된 방식에 의해 캐시나 메모리 채널의 개수에 맞게 균형 있게 할당되며 처리될 수 있다. 그러나 특정 방식으로 데이터를 처리할 때 문제가 발생할 수 있다. 예를 들면 수직으로 데이터를 처리하는 경우 특정 메모리로만 데이터가 할당되는 현상이 발생할 수 있다. 이때 트랜잭션을 할당받지 못한 메모리 구성 요소들은 쉬고 있는 유휴 (Idle) 상태가 지속된다. 이러한 경우가 지속되면 시스템은 메모리 하위시스템이 제공하는 대역폭을 얻을 수 없게 된다. 이러한 트래픽 혼잡(Traffic congestion)을 줄이기 위해 DRAM 매핑이나 정교한 인터리빙 방법이 보고되었다. 그럼에도 불구하고 기존 방식들의 문제점은 복잡한 매핑 방식이나 하드웨어 요소들이 필요하다는 점이다. 이를 해결하기 위해 단순한 매핑 방식과 최소한의 하드웨어 요소를 추가하는 것이 필요하다.

1.4 목적

1.3절에서 나타난 문제점을 해결하기 위해 복잡하지 않은 방식들이 필요하다. 기존 선형 매핑 방식의 단점을 보완하기 위해 소프트웨어 요소와 하드웨어 요소를 변경한 방식을 제안한다. 이미지와 같은 2D 데이터를 처리할 때 다중의 메모리들이 효율적으로 사용되기 위한 기술을 제안한다. 소프트웨어 요소를 변경하여 이미지를 패딩 시켜 데이터가 균등하게 메모리에 할당되도록 하였다. 또한 하드웨어 요소를 추가하여 캐시 주소를 변환해 데이터가 균등하게 메모리에 할당되도록 하였다. 각각의 소프트웨어 요소와 하드웨어 요소를 추가하거나 변경할 때 오버헤드를 최소화한다. 오버헤드를 최소화하고 효율적인 메모리 사용을 위해 조건에 따라 적응적으로 기술이 적용되도록 한다. 제안하는 각각의 기술들을 통해 부하 균형을 좋게 하여 대역폭이 향상될 수 있다. 또한 메모리가 유휴상태로 지속되는 현상을 줄어들게 하여 처리 지연 시간을 줄어들게 한다. 이를 통해 메모리 이용률을 높이는 것과 성능이 향상되는 것을 목표로 한다.

1.5 기여

본 논문의 주요 기여는 다음과 같다. 본 논문에서는 다음과 같은 특징을 갖는 적응적인 이미지 크기 패딩 기법 (pLIAM, padded Linear Address Map)과 캐시 비트 변환 기법 (CIAM, Cache Interleaved Address Map)을 제안한다. 첫째, 제시된 접근 방식은 캐시와 주 메모리 계층 구조를 모두 고려한다. 둘째, 시스템은 애플리케이션 호출 시간에 이미지 패드 크기 혹은 캐시 비트 변환을 적응적으로 결정할 수 있다. 적응형의 패드 크기 조정 알고리즘과 적응적으로 캐시 비트 변환을 진행하는 알고리즘을 개발하기 위해 메트릭 분석을 수행하고 조건을 도출한다. 또한 pLIAM과 CIAM에 대하여 각각 설계, 성능평가, 오버헤드 분석에 관해 기술한다. 실험은 제시된 설계가 트래픽 부하 균형률과 성능이 크게 향상될 수 있음을 나타낸다. 또한 분석에 따르면 이미지 크기 패딩 방식의 경우

와 이미지 크기가 큰 경우 메모리 공간 오버헤드가 중요하지 않은 것으로 나타났다. 캐시 비트 변환 과정에서의 하드웨어 오버헤드도 전체 게이트 개수에 비해 추가되는 하드웨어가 크지 않은 것으로 분석되었다.

1.6 논문의 구성

이 논문은 다음과 같이 서술된다. 제2장에서는 제시하는 설계와 관련된 연구에 관해 서술된다. 제3장에서는 제시하는 기술을 이해하기 위한 기술 배경이 서술된다. 제4장에서는 본 논문에서 제안하는 설계 방법에 관해 서술된다. 제5장에서는 그에 따른 결과와 오버헤드 측정에 관해 서술된다. 결론 및 추후 과제는 제6장에 서술된다.

제 2 장 관련 연구

2.1 DRAM 주소 매핑

DRAM의 행 버퍼 충돌을 감소시키기 위해 많은 기술이 보고되어왔다. 참고 문헌 [1]에서 XOR 연산을 통한 뱅크 인덱스 주소를 변경하는 뱅크 인터리빙 방식을 기술하였다. 참고 문헌 [2]에서 DRAM의 페이지 수를 최소화하여 행 버퍼 충돌을 줄였다. 참고 문헌 [3]에서 역방향 비트 주소 매핑을통해 메모리 액세스 대기 시간을 줄였다. 참고 문헌 [4]에서 메모리 접근 성능이 향상되기 위해 메모리 접근 흐름을 관찰하여 메모리 접근 특성을 기반으로 최적의 주소 매핑 방식을 뱅크로 전송하는 DRAM 메모리 컨트롤러인 NNAMC를 제안하였다. 참고 문헌 [5]에서 주소 매핑을 자동으로 파악하고 시각화하여 사용자에게 제공하는 DRAMDig를 제안하였다. 참고 문헌 [6]에서 DRAM의 자체 리프레시 동작 및 전력 소모를 최적화하기 위한 주소 매핑 방식에 관해 연구하였다. 참고 문헌 [7]에서 DRAM의 행 버퍼 충돌을 줄이기 위한 하드웨어와 소프트웨어 주소 매핑 기술을 제안하였다. 참고 문헌 [8]에서 버스트 (Burst) 스케줄링 접근 리오더링 (Reordering) 메커니즘을 제안하여 메모리 접근을 재배열해 데이터 버스트 간의 중첩을 가능하게 하여 메모리 대역폭을 효과적으로 활용했다. [1-8]과는 다르게 본 논문에서는 캐시와 메모리에서의 부하 균형률을 높이기 위한 픽셀과 주소 간의 매핑 방식을 보여준다.

참고 문헌 [13]에서 주소 매핑 재배열을 통해 접근 지연을 감소시키고 대역폭을 효과적으로 활용하여 DRAM의 성능을 개선하는 것을 제안하였다. 참고 문헌 [13]은 동작 중에 적응적으로 매핑 방식이 변한다는 점에서 본 논문과 유사하다. 하지만 참고 문헌 [13]과는 다르게 본 논문에서는 소프트웨어 방식도 또한 제안한다. 참고 문헌 [14]에서 동적 메모리 인터리빙을 디코더로 제어하는 방식을 제안하였다. 참고 문헌 [15]에서 랭크 액세스 패턴을 제어하기 위해 메모리 주소 매핑을 동적으로 조절하는 방식을 제안하였다. [14-15]와는 다르게 본 논

문에서는 애플리케이션이 실행 중에 적응적으로 픽셀과 주소 간의 매핑이 결정되는 기술을 보인다.

2.2 주소 생성

참고 문헌 [16]에서 DRAM 뱅크 인터리빙을 위한 하드웨어 주소 매핑 재배열 방식을 제안하였다. 참고 문헌 [16]은 본 논문의 메트릭 분석과 적응적인 주소 매핑 방식은 유사하다. 참고 문헌 [16]과는 다르게 본 논문에서는 소프트웨어 요소를 변경하는 방식이 추가되었다. 참고 문헌 [17]에서 병렬로 인터리빙 하는 아키텍처를 위한 주소 생성방식을 제안하였다. 이 방식은 통신 시스템을 위한 방식이지만 본 논문은 이미지 처리를 위한 연구라는 점에서 본 논문과 다르다. 참고 문헌 [18]에서 데이터 액세스 지역성과 충돌로 인한 캐시 미스를 줄이기 위한 루프 및 데이터 배치 변환이 제시된다. 참고 문헌 [19]에서 CUDA 기반의 전치행렬 알고리즘에서 뱅크 충돌을 패딩 없이 해결하는 방법을 제안하여 행렬 연산을 효율적으로 지원하는 방법을 제시하였다. [17-19]와는 다르게 본 논문에서는 트랜잭션 레벨의 패딩 하는 기술을 제시한다.

2.3 캐시

참고 문헌 [9]에서 메모리 요청을 우선순위에 따라 조정하여 성능을 최적화하였다. 또한 중요도 기준으로 우선순위 설정하고, 우선 처리를 보장하여 성능이 향상되는 방법을 제안하였다. 참고 문헌 [10]에서 주소를 매핑할 때 태그 필드 없이 데이터를 저장하는 방식을 제안하였다. 또한 논-캐시어블 (Non-cacheable)의 비트가 0일 때 가상-캐시 매핑이 사용되었고 캐시어블 (Cacheable) 비트가 1일 때, 가상 주소에서 물리 주소로 매핑이 사용되었다. 논-캐시어블과 캐시어블의 메모리 속성에 따라 다르게 주소가 매핑되는 것은 본

논문과 유사하다. 참고 문헌 [9-10]과 본 논문에서의 차이점은 본 논문은 캐시와 메모리 채널을 동시에 고려한 방법을 제시하고 있다는 것이다. 캐시 미스(Miss)를 줄이기 위해 참고 문헌 [11]에서는 다차원의 벡터 처리 기술을 제안하였고 참고 문헌 [12]에서는 해싱 함수를 사용하는 기술을 제안하였다. 참고 문헌 [20]에서는 여러 루프를 선택하여 표본을 뽑아 표본 기반의 매핑 방식을 제안하여 캐시 매핑을 효과적으로 하였다. [11, 12, 20]과는 다르게 본 논문에서는 부하 균형을 위한 캐시 인터리빙 방식을 제안한다.

2.4 패딩 기술

참고 문헌 [21]에서 대칭 패딩을 도입하여 컨볼루션 연산의 효율성을 높이고 CNN (Convolution Neural Network) 모델 성능이 향상되는 방법을 제안하였다. 참고 문헌 [22]에서는 작은 이미지 주위에 0으로 된 데이터로 패딩하여 컨볼루션을 진행하는 방식을 제안하였다. [21-22]와는 다르게 본 논문에서는 메모리 할당 패딩을 보여준다. 참고 문헌 [23]에서 소프트웨어 루프 반복을 위한 컴파일 타임 데이터 배치 변환 기술이 제시되었다. 이를 위해 변수 기본 주소를 조정하는 변수 간 패딩과 배열 차원의 크기를 조정하는 변수 내 패딩을 제안하였다. 참고 문헌 [24]에서 다차원 어레이의 요소 배치를 조정하여 어레이 차원의 패딩을 적용해 캐시 충돌을 최소화하였다. 참고 문헌 [25]에서 같은 배열을 공유하는 루프를 캐시 크기에 맞게 분해하고 분해된 루프를 같은 프로세서에서 연속적으로 실행하는 방식을 이용한다. 이러한 방식을 통해 다중프로세서의 매크로 작업 간의 충돌 미스를 최소화하기 위한 배열 간의 패딩을 진행한다. 참고 문헌 [26]에서 다중 레벨 캐시 충돌 미스를 줄이기 위한 패딩 기술을 제안한다. 참고 문헌 [27]에서 적분 이미지 연산을 위한 병렬 알고리즘을 제안하였다. 또한 인덱스 패딩을 진행하여 뱅크 충돌을 감소시켰다. 참고 문헌 [28]에서 SIMD (Single Instruction Multiple Data)를 이용하는 기술과 메모리 뱅크 충돌을 줄이기 위한 어레이 패딩 기술을 제시하였다. [25-28]과는 다르게 본 논문에 서는

메모리 계층에서의 부하 균형이 향상되기 위해 애플리케이션이 실행 중에 적응적으로 기술을 적용하도록 하였다.

제 3 장 기술 배경

3.1 메모리 속성

그림 3.1은 SoC 구조의 예를 나타낸다. 그림 3.1의 SoC 내부에는 네 개의 캐시와 채널이 존재하고 이미지 처리 장치나 디스플레이와 같은 입출력 장치인 마스터들이 존재한다. 마스터들은 메모리를 트랜잭션 (Transaction) 단위로 처리한다. 또한 마스터들은 읽기와 쓰기 채널로 구성되어있다. 트랜잭션에는 주소, 데이터 및 제어 정보가 포함되어있다. 이미지 픽셀이 RGB 포맷일 때 픽셀 크기는 4바이트이다. 트랜잭션의 크기가 64바이트일 때 트랜잭션은 16개의 RGB 픽셀 데이터에 접근한다. 일반적으로 트랜잭션은 메모리에 접근하는데 상당한 대기 시간이 필요한데 이에 따라 마스터는 응답이 마스터로 돌아오기 전에 여러 개의 요청을 발생한다. 이를 멀티플 아웃스탠딩 (Multiple Outstanding)이라 부른다 [29]. 이는 병렬로 일을 처리할 수 있게 하여 처리량 성능이 크게 향상되므로 최신 SoC에 널리 사용된다. 예를 들어 마스터가 4개의 요청을 발행하는 경우 멀티플 아웃스탠딩의 개수는 4개이다. 이때 해당 응답은 마스터에 도착하기 전이다.

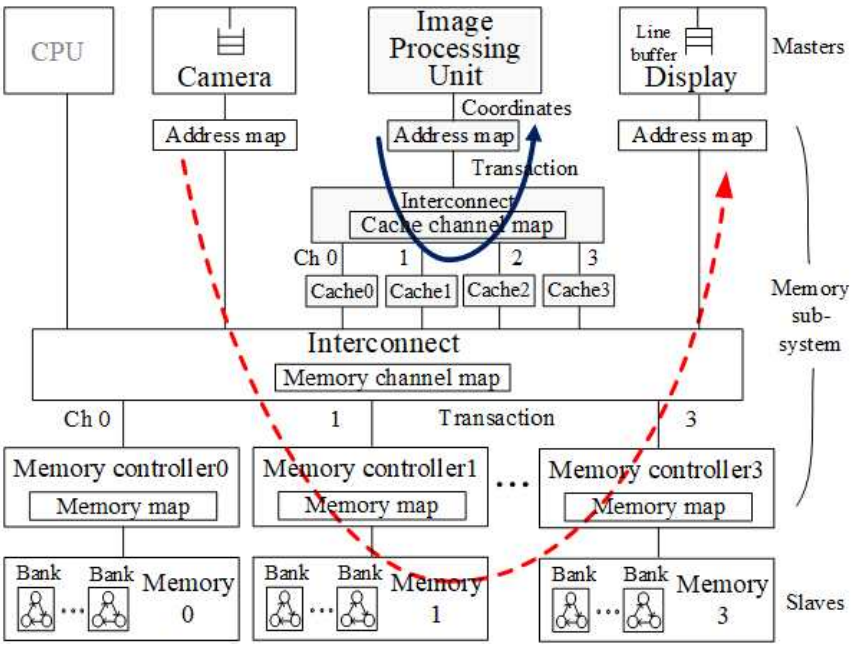


그림 3.1. Configuration of SoC.

3.1.1 캐시어블 속성

애플리케이션이 호출되면 운영체제는 메모리 공간을 할당한다. RGB 포맷의 이미지 크기가 128X32픽셀일 때 16,384(=128X32X4)바이트의 메모리가 할당된다. 애플리케이션마다 메모리 속성이 다르다, 단일 마스터가 2D 데이터 애플리케이션을 작동하고 애플리케이션에 특정 데이터 지역성이 있는 경우 운영체제는 할당된 메모리를 캐시어블 (Cacheable)로 설정한다. 이때 캐시어블은 데이터 또는 정보가 캐시에 저장할 수 있는지 아닌지를 나타내는 용어이다. 캐시는 데이터의 임시 저장소로, 주로 데이터 접근의 속도와 시스템 성능이 향상되는데 사용된다. 그림 3.1에서 이미지 처리 장치의 경우 단일 마스터로 동작한다. 이러한 경우 여러 번 액세스 될 가능성이 있는 데이터들을 캐시에 저장하여 데이터를 처리할 수 있다. 데이터를 캐시에 저장하였을 때 캐시 히트가 발생할 수 있다. 캐시 히트가 발생하면 메모리까지 거치지 않아도 되기 때문에 데이터 처리 속도를 높일 수 있다.

3.1.2 논-캐시어블 속성

캐시는 주로 데이터를 빠르게 액세스하고 성능이 향상되기 위해 사용되지만, 특정 경우에는 캐시를 사용하지 않는 것이 더 유리할 수 있다. 여러 개의 입출력 장치와 같은 하드웨어 장치와 상호작용할 때는 메모리의 데이터가 항상 최신 상태여야 하므로 캐시에 저장하지 않는다. 만약 이 마스터들이 캐시를 사용하게 된다면 어느 한 마스터는 캐시에 있는 데이터에 접근하고 다른 마스터는 메모리에 직접 접근함으로써 데이터의 일관성이 깨질 수 있다. 이러한 이유로 메모리를 논-캐시어블 (Non-cacheable) 속성으로 변경하여 캐시를 사용하지 않고 데이터에 접근하여 데이터의 일관성을 유지한다. 이 입출력 장치들은 일반적으로 메인 메모리와 직접 통신하여 데이터 일관성을 보장한다. 카메라 프리뷰 (Camera preview)의 경우 카메라 컨트롤러가 이미지를 찍고 디스플레이가 이미지를 래스터-스캔방식으로 화면에 표출한다. 이 시나리오는 그림 3.1의 점선과 같이 카메라 장치와 디스플레이 장치를 동시에 사용한다. 이런 경우 데이터의 일관성을 위해 온 칩 캐시를 사용하지 않는 논-캐시어블 속성을 채택하여 메모리에 직접 접근하여 데이터를 처리한다.

3.2 LIAM (Linear Address Mapping)

3.2.1 주소 매핑

이미지 픽셀은 고유한 주소 숫자들과 메모리의 배치에 의해 매핑된다. 주소 맵은 이미지 픽셀을 트랜잭션 주소로 변환한다. 그림 3.2는 128X32픽셀 크기의 선형 주소 맵을 나타낸다. 16픽셀을 한 묶음으로 하여 트랜잭션 하나를 구성하였다. 이러한 트랜잭션은 선형으로 순차적으로 증가하며 배치되어있다. 이는 전통적인 이미지 매핑 방식이다. 원 안의 숫자는 트랜잭션 번호를 나타내고 그 밑의 수는 16진수의 주소를 나타낸 것이다. 한 트랜잭션은 64바이트의 데이터 또는

16 RGB 픽셀을 나타낸다. 따라서 ⑨번 트랜잭션의 경우 (16, 1)의 픽셀을 나타낼 수 있다.

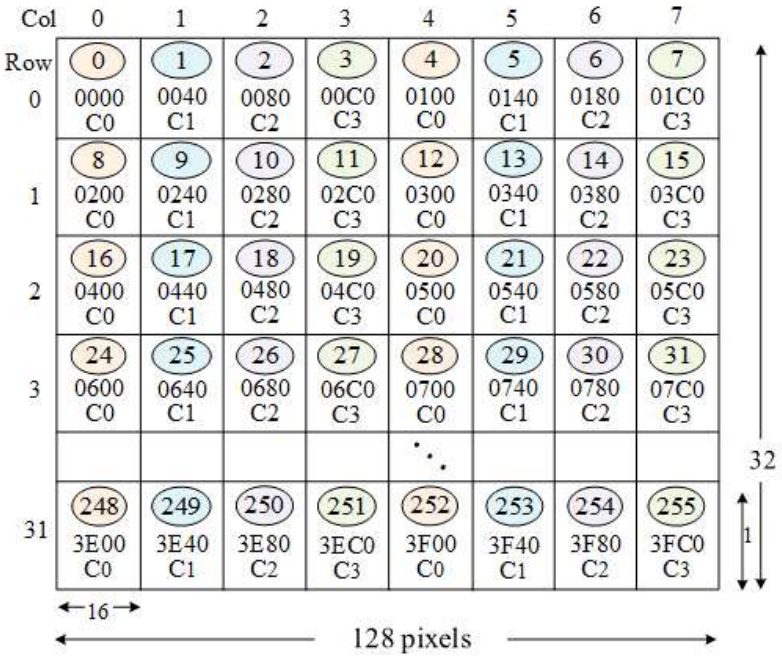


그림 3.2. Linear address map.

캐시 맵은 주소를 태그 (Tag), 인덱스 (Index), 캐시 채널 (Cache channel), 오프셋 (Offset)으로 변환한다. 그림 3.3은 캐시 라인 크기가 64바이트인 경우의 예이다. 예를 들어 트랜잭션 9의 경우 주소는 0X240이다. 0X240(0000 … 0010 0100 0000)은 캐시 채널 1로 매핑되므로 트랜잭션 ⑨는 캐시 1에 할당된다. 메모리 맵은 주소를 행, 뱅크 (Bank), 열, 메모리 채널 (Memory channel), 열로 변환한다. 예를 들어 트랜잭션 ⑨의 경우 주소는 0x240(0000 … 0010 0100 0000)이다. 이때 메모리 채널 1로 매핑된다.

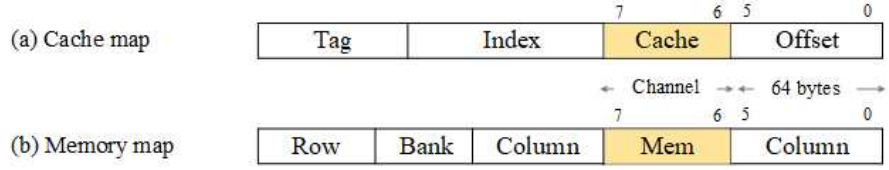


그림 3.3. Example of cache and memory mapping.

3.2.2 LIAM의 문제점

카메라 프리뷰와 같이 화면에 데이터를 그대로 표출하는 경우 보통 래스터 스캔방식을 사용한다. 래스터 스캔방식은 그림 3.4와 같이 선형으로 데이터를 처리하는 방식이다. 기존 선형 주소 맵의 경우 선형으로 데이터를 처리할 때 C0, C1, C2, C3, …. 와 같이 순차적으로 서로 다른 채널로 할당되는 것을 볼 수 있다. 여러 개의 트랜잭션이 동시에 처리 가능하여 메모리가 효율적으로 동작할 수 있다. 화면을 회전하거나 사진을 회전하는 기능을 사용하는 경우 데이터를 수직으로 처리하게 된다. 수직으로 데이터를 처리할 때 그림 3.5와 같은 방식으로 데이터를 처리하게 된다. 기존 선형 주소 맵에서 그림 3.5와 같이 데이터를 처리하는 경우 C0, C0, C0, …. 와 같이 반복적으로 한 채널에만 데이터가 할당된다. 한 채널은 동시에 한 개의 데이터만 처리할 수 있다. 그러므로 연속으로 데이터가 동일한 채널에 접근할 때 데이터 처리의 지연이 생길 수 있다. 이러한 지연은 데이터 처리 시간을 늘리고 인터리빙이 원활하게 진행되지 않아 다중의 메모리를 효율적으로 사용할 수 없게 되고 메모리 성능 저하로 이어진다.

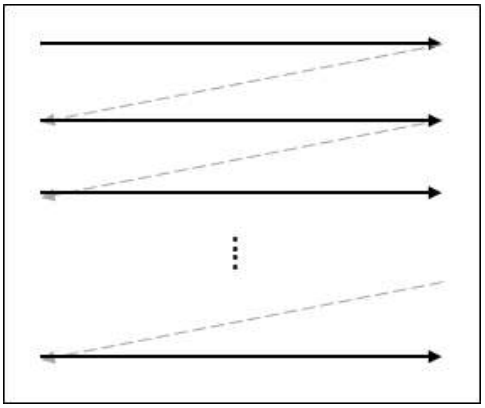


그림 3.4. Raster scan.

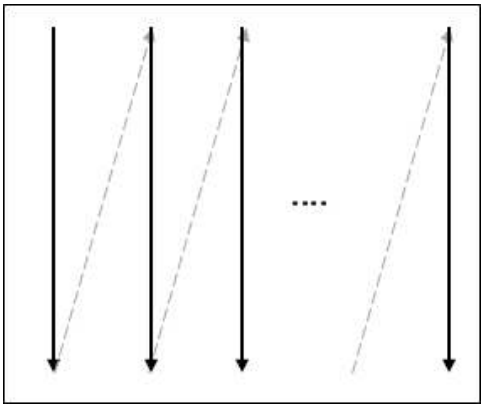


그림 3.5. Vertical scan.

3.3 DRAM

DRAM은 컴퓨터와 다른 디지털 장치에서 주로 사용되는 주기억장치 유형 중 하나이다. DRAM은 데이터를 저장하고 접근하는 데 사용되며, 컴퓨터의 작동과 데이터 처리에 필수적인 역할을 한다. DRAM의 가장 기본 단위는 메모리 셀인데 이 셀은 데이터 비트를 저장하는 데 사용된다. 각 셀은 전하를 저장하는 커패시터와 트랜지스터로 구성되어있다. 셀의 상태는 커패시터에 저장된 전하의 양에 따라 동적으로 변화된다. 이러한 셀들이 모여 뱅크를 이루고 각 뱅크들은 독립적으로 작동하며 동시에 접근할 수 있다. 데이터를 읽거나 쓸 때 원하는 뱅크를 선택하여야 하는데 뱅크는 워드 라인 (Word line)이나 비트 라인 (Bit line)을 통해 활성화되고 활성화된 뱅크의 데이터에 접근할 수 있다. 이러한 뱅크 구조는 병렬 처리를 가능하게 하여 데이터 접근 속도를 빠르게 하는 데 도움을 준다. 그러나 동시에 여러 뱅크에 접근하려는 경우 뱅크 간 충돌이 발생할 수 있다. 이는 성능 저하의 원인이 될 수 있으므로 메모리 컨트롤러 및 소프트웨어에서 효율적으로 뱅크를 관리해야 한다.

뱅크가 모여 랭크를 이루고 랭크들이 모여 DRAM을 이룬다. DRAM은 그림 3.6과 같이 메모리 컨트롤러와 채널로 연결되어 있다. 메모리 컨트롤러는 프로세서 또는 다른 장치로부터 메모리에 접근하라는 요청을 받고 이 요청은 특정 주

소로 향한다. 이 주소를 통해 메모리 위치가 식별된다. 메모리 컨트롤러는 이러한 주소를 DRAM에 보내는데 이 신호는 행 선택과 열 선택 신호로 구분된다. 행 선택 주소는 특정 행을 활성화하고, 열 선택 신호는 해당 행에서 데이트를 읽거나 쓰는 데 사용된다. 채널을 이러한 신호를 보내는 역할을 한다. 또한 DRAM은 동적 메모리로, 저장된 데이터는 시간이 지남에 따라 서서히 소멸한다. 이를 방지하기 위해 주기적으로 메모리의 모든 행을 읽어서 다시 저장하는 리프레시(Refresh) 작업이 필요하다. 메모리 컨트롤러는 이러한 리프레시 주기를 관리하고, 필요한 경우 DRAM을 갱신한다. 또한 다른 시스템 구성 요소와 메모리 간의 효율적인 데이터 전송을 조정한다. 이를 통해 데이터를 안정적으로 읽고 쓸 수 있게 하고, 시스템 성능을 개선할 수 있다.

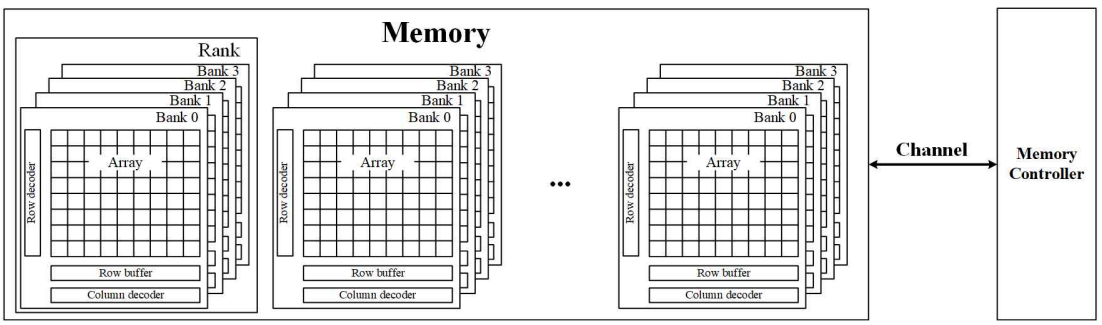


그림 3.6. DRAM architecture.

3.4 Cache

캐시는 메인 메모리와 CPU (Central Processing Unit) 간 데이터 처리 속도 향상을 위해 중간에서 일시적으로 데이터를 보관하는 메모리이다. 용량은 적지만 속도가 빨라 데이터에 접근하기에 용이하다. 자주 쓰이는 데이터나 프로그램 명령을 캐시에 저장하면 CPU와 같은 장치들이 명령을 내릴 때 메인 메모리로 가기 전 캐시로 먼저 접근하여 필요한 데이터를 사용할 수 있게 한다. 이러한 캐시는 L1, L2, L3의 레벨로 나뉜다. L1은 CPU와 같은 장치들에 가장 가까이에 있

어 속도가 가장 빠르고 숫자가 커질수록 CPU와 멀어지며 속도도 느려진다. 캐시는 공간적 지역성과 시간적 지역성을 가지고 있는데 지역성 별 특징은 다음과 같다.

- 공간적 지역성 (Spatial locality) : 공간적 지역성은 한 번 참조한 메모리의 옆에 있는 메모리를 다시 참조하게 되는 성질이다. 특정 데이터와 가까운 주소가 순서대로 접근되었을 경우 한 메모리 주소에 접근할 때 그 주소뿐만 아니라 해당 블록을 전부 캐시에 가져와 캐시 효율성이 높아진다.
- 시간적 지역성 (Temporal locality): 시간적 지역성은 한 번 참조된 주소의 내용은 곧 다음에 다시 참조되는 특성이다. 메모리상의 같은 주소에 여러 차례 읽기 쓰기를 수행할 경우인데 이는 상대적으로 작은 크기의 캐시를 사용해도 효율성이 높아진다.

이러한 지역성을 가진 데이터가 캐시에 저장될 때 캐시가 효율적으로 동작하여 캐시의 적중률 (Hit rate)을 극대화할 수 있다.

캐시에 데이터의 메모리 주소가 접근할 때 메모리 주소를 이용하여 찾고 있는 데이터가 캐시 블록에 있는지 확인한다. 캐시 메모리에 찾는 데이터가 존재하였을 때 캐시 히트 (Hit) 라 나타내고, 캐시 메모리에 찾는 데이터가 존재하지 않을 때 캐시 미스 (Miss)라고 나타낸다. 캐시 미스가 발생하면 메인 메모리 저장소로부터 필요한 데이터를 찾아 캐시 메모리에 불러온다. 메모리 주소는 캐시 태그 (Tag), 세트 인덱스 (Set index), 블록 오프셋 (Block offset)으로 구성된다. 이 구성은 메모리 주소가 캐시 블록을 식별할 수 있도록 한다. 캐시 태그 부분은 캐시의 태그 필드 값과 비교하는 데 사용된다. 세트 인덱스는 블록을 선택하는 데 사용된다. 블록 오프셋은 캐시 블록 내의 특정 바이트 위치에 접근하기 위한 부분이다. 이 주소의 최하위 비트 부분인 오프셋 필드는 블록 내의 워드를 선택할 때 사용하지 않는다. 캐시 히트와 미스를 판별하는 과정은 다음과 같다. 주소의 인덱스와 캐시 엔트리가 선택된다. 이후 주소의 태그와 캐시 엔트리의 태그가 일치할 때 캐시가 적중되어 데이터를 프로세서에 전달한다. 주소의 태그와 캐시 엔트리의 태그가 일치하지 않을 때 적중되지 않아 캐시 미스가 발생한다. 이에 따라 메인 메모리로 접근하게 된다. 이 과정은 그림 3.7에 묘사되어 있다.

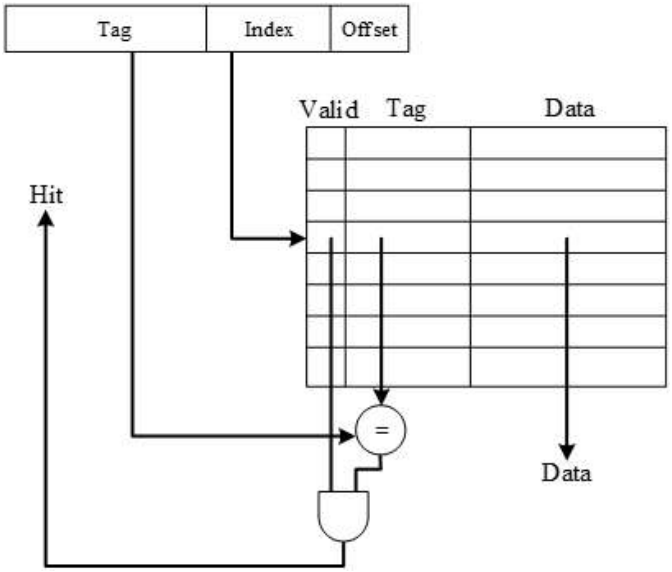


그림 3.7. Hit and miss operation in a cache.

3.5 AXI 프로토콜

AXI는 Advanced eXtensible Interface의 약어로 IP (Intellectual Property) 간의 데이터 전송을 위한 인터페이스 규약이다. AXI 프로토콜은 ARM에서 정의한 표준 온-칩 인터커넥트(버스) 프로토콜이다[29]. AXI의 특징은 높은 대역폭, 파이프라인화된 동작 (Pipelined operation), 버스트 단위의 전달, 멀티플 아웃스탠딩 리퀘스트 (Multiple outstanding request), 아웃 오브 오더 (Out-of-order) 트랜잭션 완료 지원이다. AXI는 그림 3.8과 같이 읽기와 쓰기를 구분하여 다섯 개의 채널로 분리되어 있다. 또한 각 채널들은 독립적으로 동작한다. 쓰기 상황에서 마스터가 슬레이브에 데이터를 전송하면 마스터 입장에서는 슬레이브가 데이터를 전부 받았는지 알 수 없다. 데이터가 전송 중에 다른 행동으로 인한 에러가 발생하지 않도록 응답 채널이 별도로 필요하다. 또한 읽기 상황에서는 마스터가 슬레이브에 데이터를 요청하면 슬레이브가 마스터에게 데이터를 전달해주는 과정이다. 데이터를 모두 전달하면 더 이상 데이터를 전달하지 않는다는 의미 자체가 완료했다는 신호이므로 응답 채널이 존재하지 않는다.

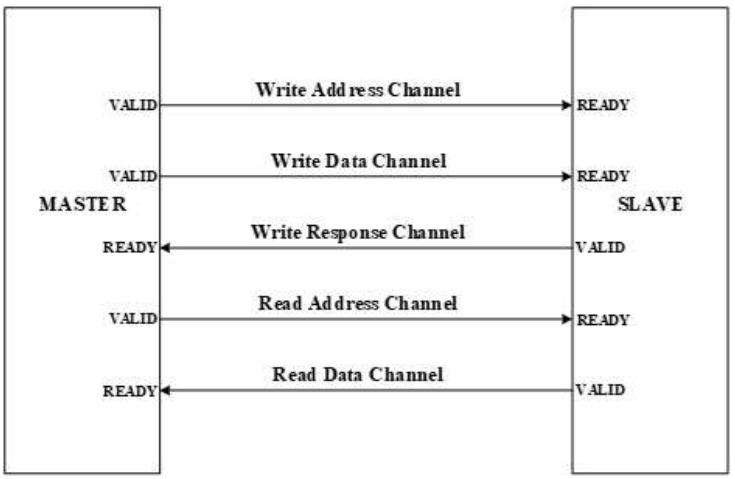


그림 3.8. Channel construction of AXI protocol.

AXI 버스는 Valid-Ready 핸드쉐이크 (Handshake)를 사용한다. 이는 각각의 채널마다 필요하다. Valid는 데이터를 보내는 쪽에서 보낼 준비가 되었다는 신호를 나타내고 Ready는 데이터를 받는 쪽에서 받을 준비가 되었다는 신호를 나타낸다. 어느 한쪽에서 Valid 신호를 보내고 반대쪽에서 Ready 신호를 보내면 그 때 핸드쉐이크가 이루어져 데이터가 전송된다. 그림 3.8에서와 같이 Valid와 Ready는 어느 마스터나 슬레이브에 국한되는 것이 아닌 데이터를 받는 쪽에 따라 달라진다.

AXI 프로토콜과 기존 프로토콜의 차이점은 단일 파이프라인으로 주소와 데이터가 하나로 되어 있어 한 개의 주소로 한가지의 데이터만 처리할 수 있었다. 그러나 AXI 프로토콜의 경우 한 개의 주소로 여러 개의 데이터를 동시에 처리할 수 있게 되었다. 이를 버스트 (Burst)라 표현한다. AXI 프로토콜의 경우 읽기와 쓰기를 동시에 할 수 있다. 또 다른 특성으로는 아웃 오브 오더 트랜잭션 완료 지원이 있는데 아웃 오브 오더는 미완성, 미해결된 이라고 해석할 수 있다. 기존 프로토콜의 경우 미해결된 트랜잭션이 존재할 때 다음 데이터를 처리하지 못하였다. 이 미해결된 트랜잭션이 쌓일 때 데이터 처리가 지연된다. 이와는 다르게

AXI 프로토콜은 멀티플 아웃스탠딩 리퀘스트와 이 특성으로 미해결된 트랜잭션이 존재하더라도 그다음 트랜잭션이 데이터 처리가 될 수 있는 경우 먼저 처리될 수 있다. 이러한 특성을 통해 데이터 처리 지연 시간을 줄이고 빠르게 처리할 수 있다.

제 4 장 제안하는 설계

4.1 pLIAM (padded Linear Address Map)

기존 LIAM의 특정 부하로만 데이터가 처리되는 현상을 해결하기 위해 pLIAM은 소프트웨어 요소 (이미지 크기)를 변경하여 데이터 매핑 방식을 변환하였다. pLIAM은 이미지 크기를 패딩하는 방식으로 기존의 방식과는 다르게 메모리 채널과 캐시 채널을 둘 다 고려하여 효율을 개선한 방식이다. 이 방식은 기존 이미지 맵에서 캐시어블의 마스터일 때 캐시 라인 크기로 혹은 논-캐시어블의 마스터일 때 한 트랜잭션 크기로 맵을 확장하여 수직으로 데이터가 처리할 때 연속해서 다른 캐시가 할당되도록 하였다. 그림 4.1에서 주소 매핑 예시를 볼 수 있다. 한 트랜잭션의 크기는 16 RGB 픽셀 혹은 64바이트이다. 이미지 가로 크기가 128픽셀인 경우 그림 4.1의 LIAM과 같이 이미지를 가로로 처리할 때 C0, C0, C0, … C0, C1, C1, …. 와 같이 연속적으로 같은 채널 혹은 캐시로 할당되는 것을 볼 수 있다. 이때 한 트랜잭션의 크기만큼 이미지를 패딩을 수행할 수 있다. 패딩 한 결과 이미지 크기가 128픽셀에 16픽셀이 더해져 이미지 가로 크기가 144픽셀이 된다. 이러한 과정을 통해 pLIAM과 같은 결과를 얻을 수 있다. 이를 통해 그림 4.1의 pLIAM과 같이 데이터가 수직으로 처리될 때 C0, C1, C2, C3, …. 와 같이 연속적으로 다른 채널 혹은 메모리에 할당되는 것을 볼 수 있다. 이것은 pLIAM이 LIAM보다 인터리빙과 부하 균형이 향상되는 것을 의미한다. 그러나 이미지 가로 크기에 따라 수직으로 데이터를 처리할 때 연속으로 서로 다른 메모리에 할당되는 경우가 있다. 이러한 경우에는 패딩을 적용하지 않는 것이 유리하다. 따라서 이 패딩 방식을 이미지 가로 크기에 따라 적응적으로 실행할 것이다. 이 적응적으로 진행하기 위한 메트릭 분석은 4.3절에서 분석한다.

패딩을 진행할 때 패딩 된 부분은 데이터를 처리하는 데 사용되지 않지만, 메모리는 할당된다. 이때 LIAM과 비교하여 메모리 공간이 더 필요하다. 그림 4.1

의 경우 이미지의 세로 크기 32, 이미지의 가로 크기 128픽셀, RGB일 때 4바이트이다. 이 셋을 모두 곱하면 16,384바이트가 된다. pLIAM의 경우 이미지의 가로 크기 144, 이미지의 세로 크기 32, RGB일 때 4바이트이다. 이 셋을 모두 곱하면 18,432바이트가 된다. LIAM의 경우 16,384바이트만큼의 메모리 공간이 필요하고 pLIAM의 경우 18,432바이트만큼의 메모리 공간이 필요하다. pLIAM을 적용할 때 추가로 2,048바이트, 대략 13%만큼의 오버헤드가 발생한다. 이 오버헤드는 이미지 크기마다 증가율이 달라지는데 이는 5.4절에서 언급할 것이다.

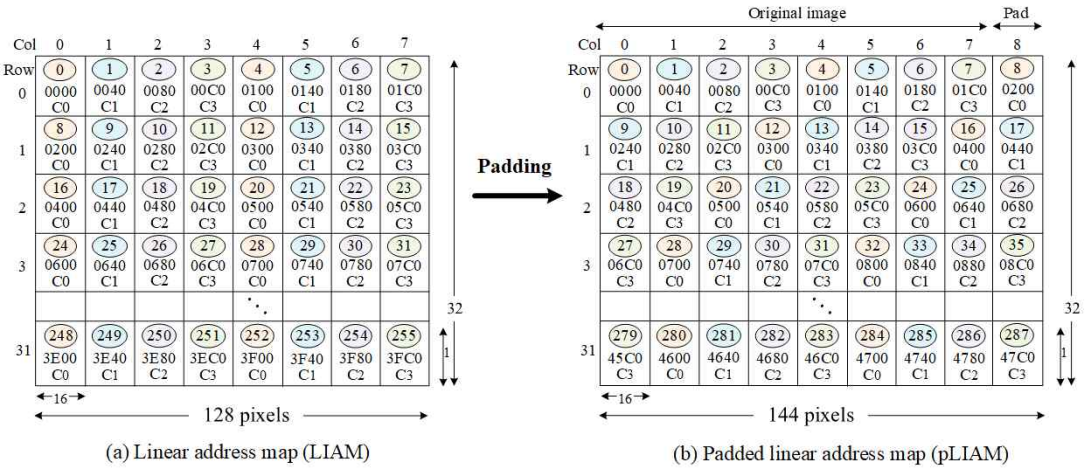


그림 4.1. Comparison of LIAM and pLIAM.

4.2 CIAM (Cache Interleaved Address Map)

캐시어블의 유닛이 데이터를 처리할 때 기존 주소 맵의 경우 수직으로 처리할 때 하나의 캐시로만 데이터가 할당되는 경우가 있다. 이러한 문제를 해결하기 위해 기존 선형 주소 맵에서 하드웨어 요소를 추가하였다. CIAM은 기존 주소 맵에서 한 트랜잭션의 캐시 비트와 인접한 트랜잭션의 캐시 비트를 변환하여 주소 매핑 방식을 변경한다. 그 예시는 그림 4.2와 같다.

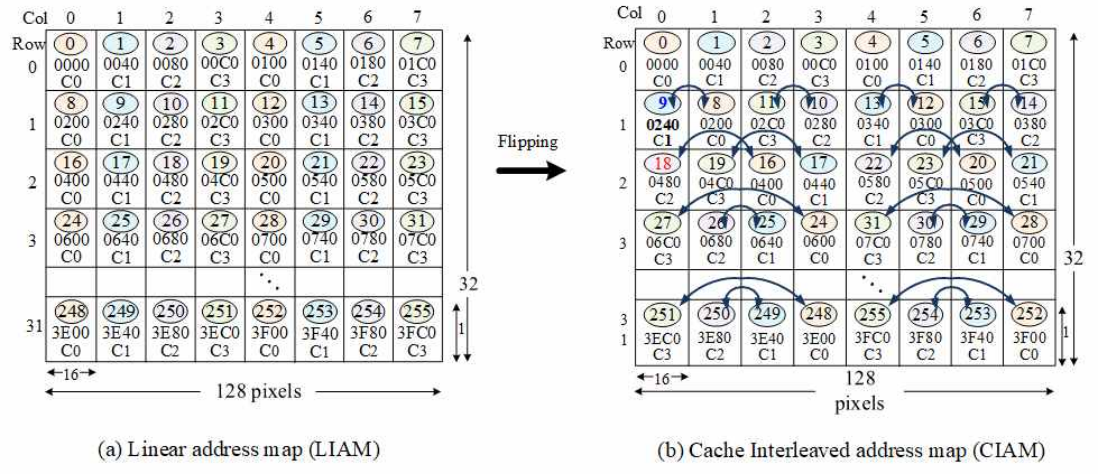


그림 4.2. Comparison of LIAM and CIAM.

캐시 비트를 변환하는 데 필요한 구성 요소들이 표 4.1에 나타나 있다. 캐시 비트를 변환할 때 수직으로 데이터를 처리할 때 데이터의 첫 줄인 행 0은 비트 변환을 하지 않는다. 행 1의 경우 C0와 C1, C2와 C3를 변환한다. 행 2의 경우 C0와 C2, C1과 C3를 변환한다. 행3의 경우 C0와 C3, C1과 C2를 변환한다. 이러한 패턴으로 반복하여 변환한다면 그림 4.3과 같이 수직으로 데이터를 처리할 때 C0, C1, C2, C3, … 이러한 순서로 캐시가 할당된다. 변환할 트랜잭션은 *GSLN*에 의해서 결정된다. *GSLN*을 캐시 개수로 나눈 후의 나머지와 변환하고자 하는 트랜잭션의 캐시 비트를 XOR (Exclusive or) 하면 해당 LIAM 트랜잭션의 캐시 비트가 변환된다. 이 과정은 그림 4.3에 과정과 예시가 나타나 있다.

표 4.1. CIAM configuration.

ImgHB	Image horizontal size	*ImgHB* = ImgH X BytePixel
NumCache	Number of caches	
SLS	Super-line size (in bytes)	*SLS* = (Number of caches) X (Line size)
SLN	Super-line number	*SLN* = LIAM address / *SLS*
GSLS	Grouped super-line size (in bytes)	*GSLS* = *k* X *SLS*
GSLN	Grouped super-line number	*GSLN* = floor (*SLN* / *k*)
k	1 If *ImgHB* < *SLS* round(*ImgHB*/*SLS*) Otherwise	*k* indicates how many superlines that a grouped superline has.
T	*ImgHB*/*SLS*	*T* indicates how many superlines are associated with an image row.

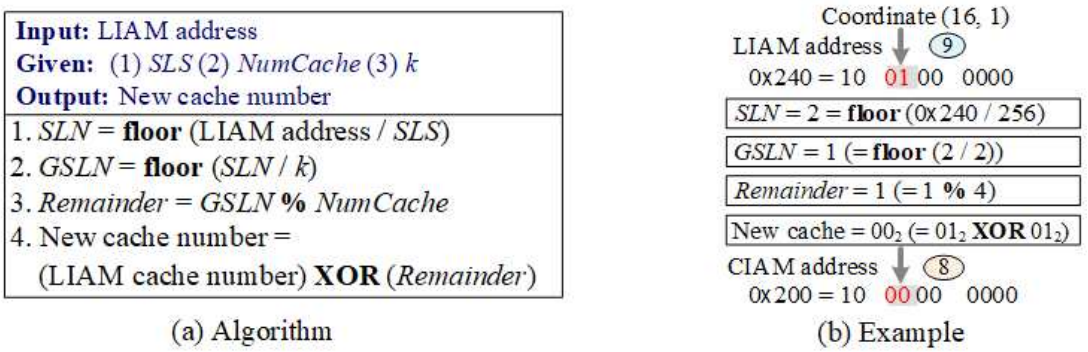


그림 4.3. Cache flipping procedure and the flipping example.

이러한 과정에서 추가의 하드웨어 장치가 필요하다. 그림 4.4는 CIAM의 하드웨어 로직 컨버터를 나타낸다. 그림 4.4에서 캐시 플립핑 (Cache flipping) 부분에는 그림 4.3 (a)의 알고리즘이 적용된다. 캐시 플립핑 부분에서 CIAM의 주소 맵을 얻을 수 있다. 또한 모든 경우에 CIAM이 유리한 것은 아니기 때문에 CIAM이 적용되어야 할 상태를 확인하는 것은 컨디션 체커 (Condition checker)에서 판단된다. 컨디션 체커에서 조건을 확인한 후 조건에 맞았다면 1을 선택하여 CIAM이 적용될 것이고 그렇지 않았을 때 0이 선택되어 LIAM이 적용될 것이다. CIAM이 적용되어야 할 상태는 4.3절에서 분석한다.

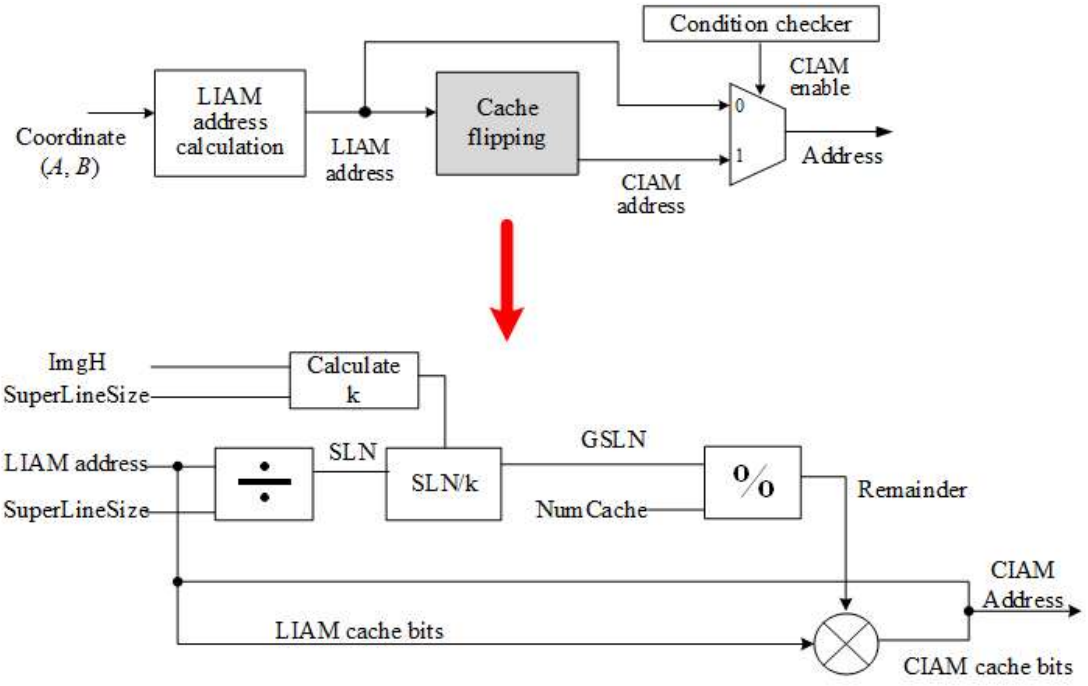


그림 4.4. CIAM hardware logic converter.

4.3 메트릭 분석

제안한 기술들이 모든 경우에 적용되는 것이 도움이 되는 것은 아니다. 다음과 같은 경우들은 새로운 방식을 적용하지 않아도 되는 경우이다.

- 선형으로 데이터가 처리되는 경우 기존 주소 매핑 방식과 제안한 주소 매핑 방식의 패턴이 비슷하므로 선형으로 데이터를 처리할 때 기존 주소 매핑 방식을 적용하는 것이 유리하다.
- 이미지 크기가 작은 경우 제안한 방식을 적용할 때 실행시간은 줄어들 수 있다. 그러나 그에 대한 오버헤드 발생이 크고 또한 기존 선형 매핑을 사용하더라도 실행 사이클이 그렇게 크지 않으므로 기존 주소 매핑 방식을 적용하는 것이 유리하다.
- 특정 이미지 크기에 따라 기존 선형 주소 맵에서 수직으로 데이터를 처리할

때 동시에 여러 메모리에 할당되는 경우가 있다. 이러한 경우 제안한 방식을 적용하지 않아도 된다.

또한 pLIAM의 경우 메모리가 캐시어블인 경우 메모리가 캐시를 사용하므로 패딩을 캐시 라인 크기로 진행하여야 하고 논-캐시어블인 경우 패딩을 트랜잭션 크기로 진행해야 한다. CIAM의 경우 캐시 주소를 변환하기 때문에 CIAM은 캐시어블 인 경우에는 변환을 진행하고 논-캐시어블인 경우 변환을 적용하지 않는다.

4.3.1 메트릭 수식 분석

제시한 기술들을 적용하였을 때 이득이 되는 이미지 크기가 있고 그렇지 않은 크기가 있다. 이미지의 가로 크기에 따라 한 행의 트랜잭션 개수가 다르다. 이 개수가 SLS의 배수일 때 혹은 $\frac{SLS}{2}$의 배수일 때 수직으로 데이터를 처리하는 경우에만 제시한 기술을 적용하는 것이 유리하다. 그러나 그렇지 않았을 때 제안하는 기술을 적용하지 않는 것이 유리하다. 이때 제안하는 방법을 적용해야 할 조건을 알아보기 위한 메트릭을 분석하였다.

$$Metric = \begin{cases} 1, & \text{인접한 트랜잭션이 다른 메모리에 접근} \\ 0, & \text{인접한 트랜잭션이 같은 메모리에 접근} \end{cases} \tag{4.1}$$

주위 트랜잭션과 비교하여 서로 다른 트랜잭션이 다른 메모리에 접근하는 경우 0으로 할당하고 같은 메모리에 접근하는 경우 1로 할당한다. 이는 식 4.1과 같다. 할당된 모든 메트릭을 모두 더한 후 이와 같은 과정을 트랜잭션이 동, 서, 남, 북 방향으로 반복한다. 이 과정은 식 4.2와 같다.

$$Metric_{i,j} = \sum_{N,S,E,W} \sum_{m=1}^{M-1} Metric_{adj} \tag{4.2}$$

$$Average\ metric = \frac{Metric_{i,j}}{Total\ nmber\ of\ transactions} \quad (4.3)$$

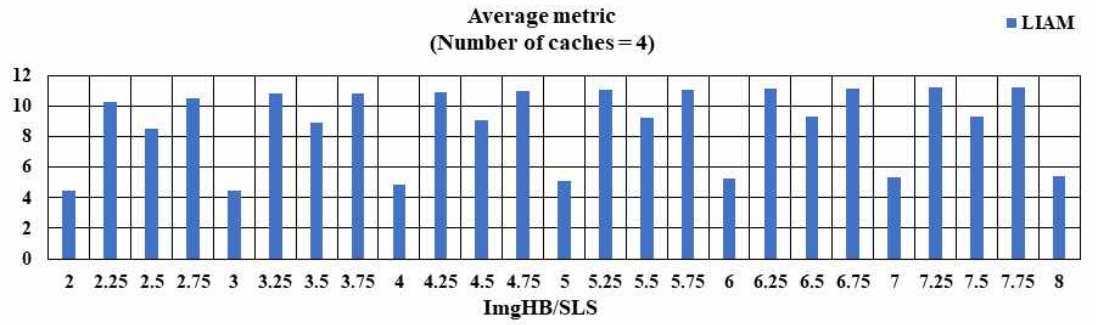


그림 4.5. Average metric of LIAM.

평균 메트릭을 구하기 위해 메트릭 들을 모두 더한 후 식 4.3과 같이 이를 트랜잭션의 개수로 나눈다. 평균 메트릭을 그래프로 나타낸 결과 그림 4.5와 같은 그래프를 얻을 수 있다. 평균 메트릭이 클수록 서로 인접한 트랜잭션이 서로 다른 메모리에 접근하고 있음을 의미한다. 평균 메트릭이 작을수록 트랜잭션에 서로 같은 메모리에 접근하는 트랜잭션이 많다는 것을 의미한다. 평균 메트릭이 작은 부분을 분석한 결과 식 4.4를 얻을 수 있었다.

$$\frac{p}{2} - 0.125 \le \frac{ImgHB}{SLS} \le \frac{p}{2} + 0.125 \qquad (p\text{는 정수}) \quad (4.4)$$

식 4.4에 만족하는 경우는 평균 메트릭이 작은 경우로 제시된 기술을 적용할 수 있다. *ImgHB*는 이미지의 가로 픽셀 크기와 설정 포맷(RGB, YUV….)의 픽셀 크기를 곱한 값이다. 이 수식은 이미지 가로 크기에 따라 결정된다. 식 4.4에서 *ImgHB*를 *SLS*로 나눈 값을 *T*라고 표현한다. 이를 식 4.5에 표현하였다.

$$T = \frac{ImgHB}{SLS} \quad (4.5)$$

식 4.5는 이미지의 한 행에 몇 개의 캐시 혹은 채널 들이 연관되어있는지를 표시한다. 실험에서 모든 이미지 크기에 대한 T값을 구한 후에 식 4.4와 비교한 후 CIAM 혹은 pLIAM의 적용 여부를 확인한다.

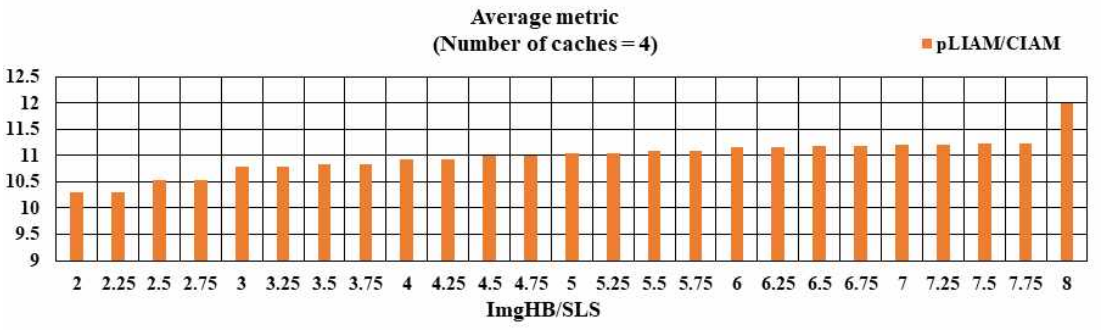


그림 4.6. Average metric of pLIAM or CIAM.

그림 4.5는 T값이 식 4.4에 만족하는 경우 pLIAM 혹은 CIAM을 적용한 경우이다. 그림 4.4과 그림 4.5를 비교하였을 때 그림 4.5에서 pLIAM 혹은 CIAM을 적용하였을 때 평균 메트릭의 수가 증가한 것을 확인할 수 있다. 이를 통해 인접한 트랜잭션들이 서로 다른 채널에 할당될 수 있음을 알 수 있다.

4.3.2 메트릭 적용 알고리즘

앞서 언급한 조건들을 이용하여 적응적으로 기법을 적용한 알고리즘이 그림 4.6에 있다. 그림 4.6에서 데이터를 처리할 때 선형패턴일 때 혹은 이미지 크기가 매우 작을 때 기법을 적용하지 않는다. 또한 식 4.4에 만족하지 않는 경우 제시된 방식을 적용하지 않는다. 그러나 앞선 상태의 반대 경우에서 메모리의 속성이 캐시어블인 경우 pLIAM에서는 라인 크기만큼 패딩을 진행하고 캐시어블이 아닌 경우 트랜잭션 크기만큼 패딩을 진행한다. 또한 CIAM의 경우 캐시어블인 경우 캐시 비트를 변환하고 그렇지 않았을 때 변환을 진행하지 않는다. 이러한

조건들을 통해 적응적으로 기법을 적용하여 더욱 불필요한 메모리 오버헤드 또는 하드웨어 오버헤드를 줄일 수 있다.

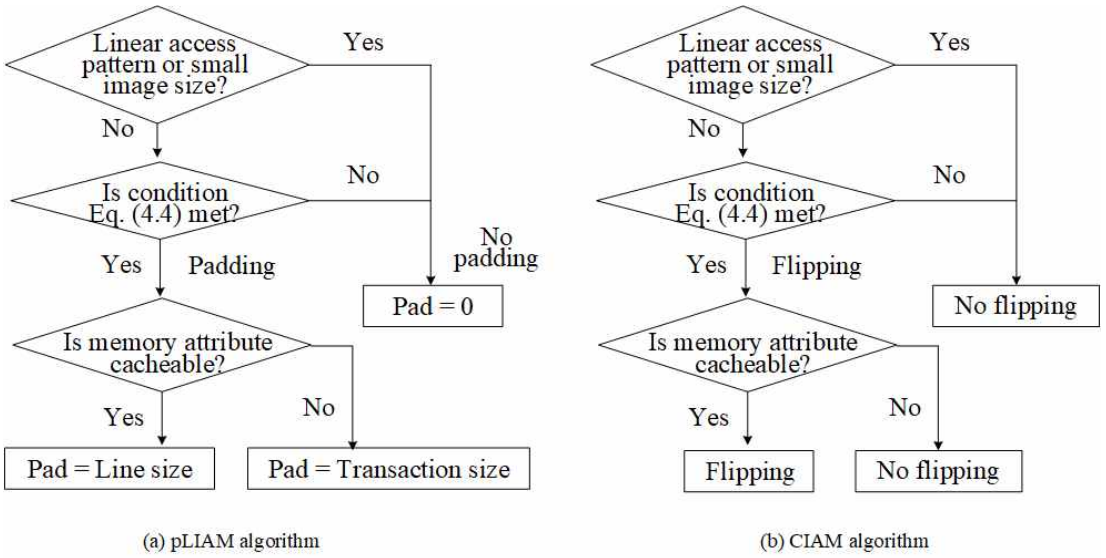


그림 4.7. Proposed pLIAM and CIAM algorithm.

제 5장 실험 결과

5.1 시스템 구성

제시한 기술에 대해 실험하기 위해 시스템 구성을 표 5.1과 같이 하였다. 구성 요소들은 AXI 버스 프로토콜을 사용하여 서로 통신한다[29]. 패드 크기를 적용하는 알고리즘을 C++으로 구현하였고 이를 시스템 모델에 통합하였다. 또한 SoC 구조는 그림 3.1에 묘사되어 있다.

표 5.1. System setting configuration.

Components	Item	Configuration
Cache	Channels	Configurable
	Line size	64 bytes
	Organization	16-way set associative
	Mapping	Tag, Index, Channel, Offset
	Size	512 lines
	Replacement	Least Recently Used(LRU)
Interconnect	Data width	128 bits
	Arbitration	Round-Robin
	Transaction size	64 bytes
	Multiple outstanding	Max 16
Memory Controller	Mapping	Row, Bank, Col, Channel, Col
	Request queue	16 entries
Memory (DRAM)	Model	DDR3-800
	Timing	$t_{CL}-t_{RCD}-t_{RP}=5-5-5$
	Channels	Configurable
	Scheduling	bank-hit first
	Banks	4

표 5.2는 워크로드 (Workload) 시나리오를 보여준다. 이미지를 화면에 표출하는 카메라 프리뷰 (Camera preview), 이미지의 크기를 1.5배로 확대하는 이미지 스케일링 (Image scaling), 두 개의 이미지를 합치는 이미지 블렌딩 (Image blending)의 경우 래스터-스캔방식으로 데이터를 처리하기 때문에 패딩 크기는 0이다. 또한 논-캐시어블이므로 CIAM은 적용되지 않는다. 회전된 이미지를 화면으로 표출하는 로테이티드 디스플레이 (Rotated display)와 카메라로 이미지를 찍고 회전된 이미지를 화면으로 표출하는 로테이티드 프리뷰 (Rotated preview)의 경우 데이터를 수직으로 처리하는 과정이 포함되므로 pLIAM이 적용되어 패딩 크기는 0 또는 트랜잭션 (Transaction) 크기가 된다. 두 과정의 경우도 논-캐시어블이므로 CIAM은 적용되지 않는다. 영상의 윤곽선을 검출하는 데 사용되는 엣지 디텍션과 CNN에서 사용되는 합성곱 연산인 컨볼루션의 경우 블록 단위로 수직으로 처리되는 과정이 포함되어있다. 그러므로 패딩 크기는 0 또는 캐시라인 크기가 된다. 두 워크로드의 경우 캐시어블이기 때문에 CIAM도 또한 적용될 수 있다.

표 5.2. Workload (NC is non-cacheable, C is cacheable).

Workloads	Type	Component	Operation	Access	Pad Size	CIAM
Camera preview	NC	Camera	Write	Raster-scan	0	X
		Display	Read	Raster-scan		
Image scailingX1.5	NC	Camera	Write	Raster-scan	0	X
		Scaler	Read, Write	Raster-scan		
		Display	Read	Raster-scan		
Image blending	NC	Blender	Read, Read, Write	Raster-scan	0	X
Rotated display	NC	Display	Read	Vertical	0 or TransSize	X
Rotated preview	NC	Camera	Write	Vertical	0 or TransSize	X
		Display	Read	Raster-scan	0 or TransSize	X
Edge detection	C	Image processing unit	Read, Read, Write	Block	0 or LineSize	O
Convolution	C	Image processing unit	Read, Read, Write	Block	0 or LineSize	O

5.2 실행 사이클 측정

실행 사이클은 데이터가 처리되는 사이클 수를 측정한 것이다. 실행 사이클 수가 클수록 데이터 처리 과정에서 지연이 많이 됨을 뜻하고 그러므로 실행 사이클 수가 적을수록 데이터가 처리 도중에 지연이 줄어들었다는 의미로 메모리가 효율적으로 동작함을 의미한다. 실험을 진행하기에 이미지 크기에 따른 *T*값을 먼저 구하였다. *T*값은 *ImgHB*를 *SLS*로 나눈 것이다. 이때 *ImgHB*의 경우 이

미지의 가로 크기와 바이트 픽셀 수를 곱한 값이고 *SLS*는 캐시 혹은 메모리의 개수와 라인 크기를 곱한 값이다. 바이트 픽셀의 경우 RGB를 대상으로 실험하였기 때문에 4바이트 픽셀이 되고 라인 크기는 한 트랜잭션당 16픽셀에 4(RGB 픽셀)가 곱해져 64바이트가 된다. 실험에서 사용될 이미지 크기와 이미지 크기에 따른 *T*값과 그에 따른 알고리즘 적용 여부를 표 5.3에 나타내었다.

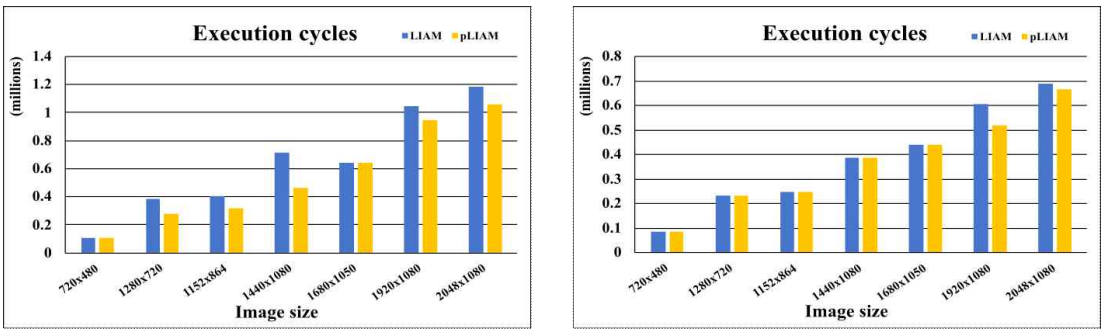
표 5.3. *T* value analysis.

Image Size(Pixels)	$T = \frac{ImgHB}{SLS}$	Equation (4.4)
	4 Channels	4 Channels
720X480	11.25	Not met
1280X720	20.00	Met
1152X864	18.00	Met
1440X1080	22.50	Met
1680X1050	26.25	Not met
1920X1080	30.00	Met
2048X1080	32.00	Met

5.2.1 pLIAM의 실행 사이클

로테이티드 디스플레이 (Rotated display)와 로테이티드 프리뷰 (Rotated preview)는 수직으로 데이터를 처리하는 과정이 포함되어있어 pLIAM을 적용할 수 있다. 그 결과 식 4.4조건에 만족하는 경우 pLIAM이 적용되었다. 그림 5.1에서 (a)의 경우 식 4.4에 만족하는 이미지 크기인 1280X720, 1152X864, 1440X1080, 19020X1080, 2048X1080은 표 5.3에 따라 *T*값이 식 4.4에 만족하므로 pLIAM이 적용되어 실행 사이클의 수가 줄어든 것을 그림 5.1에서 확인할 수 있다. 이때 평균 11.9%, 최대 34.8%의 감소율을 확인할 수 있었다. 나머지 이미지 크기의 경우 식 4.4에 만족하지 않아 패드 크기가 0으로 적용되어 실행 사이클이 그대로인 것을 확인할 수 있다. 그림 5.1에서 (b)의 경우 식 4.4에 만족하는 경우 pLIAM이 적용되어 실행 사이클이 줄어드는 효과를 기대하였으나 1280X720, 1152X864, 1440X1080의 이미지 크기에서 실행 사이클 수가 비슷

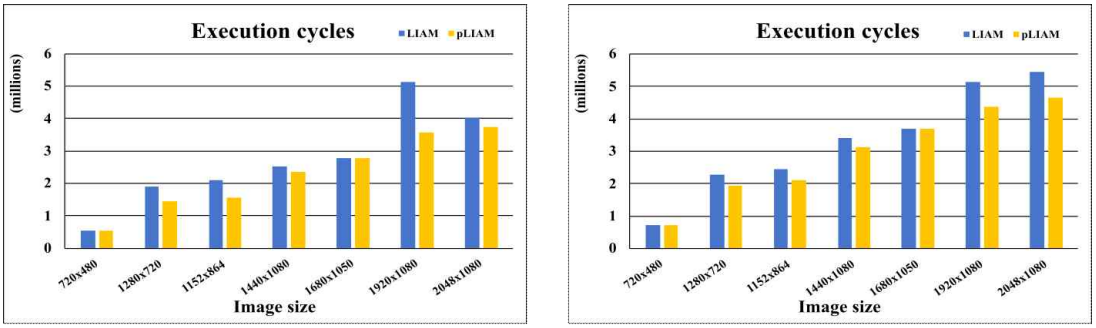
한 결과를 얻을 수 있었다. 이는 분석을 통해 다수의 마스터가 동시 동작 도중 메모리의 뱅크 충돌에 의해 실행 사이클이 지연되는 것을 확인할 수 있었다. 식 4.4에 만족하는 다른 이미지 크기인 1920X1080, 2048X1080의 경우 실행 사이클이 줄어든 것은 확인할 수 있다. 이때 평균 8%, 최대 14.4%의 감소율을 확인할 수 있었다. 마찬가지로 나머지 크기의 데이터의 경우 pLIAM이 적용되지 않아 실행 사이클 수가 같은 것을 알 수 있다.



(a) Rotated display (b) Rotated preview

그림 5.1. Execution cycles of non-cacheable attribute in pLIAM (4 caches, 4 channels).

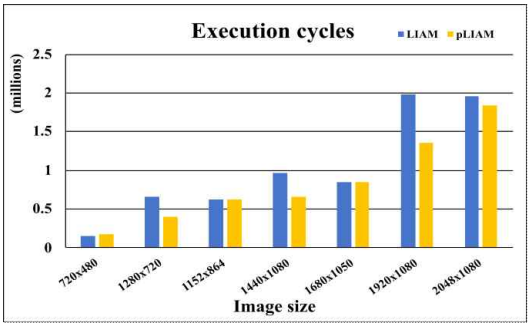
그림 5.2의 경우 컨볼루션 (Convolution)과 엣지 디텍션 (Edge detection)의 실행 사이클의 결과를 나타낸다. 컨볼루션과 엣지 디텍션 연산은 블록 단위로 데이터를 처리하는 과정에서 데이터를 블록 단위 내에서 수직으로 처리한다. 그러므로 *T*값이 식 4.4에 만족하는 이미지 크기인 1280X720, 1152X864, 1440X1080, 1920X1080, 2048X1080에 pLIAM이 적용될 수 있다. pLIAM을 적용한 결과 컨볼루션의 경우 평균 9.7%, 최대 30.1%의 감소율을 보였다. 또한 엣지 디텍션의 경우 평균 7.4%, 최대 14.6%의 감소율을 보였다. *T*값이 식 4.4에 만족하지 않는 경우의 이미지 크기인 720X480, 1680X1050의 이미지 크기에서는 패드 크기가 0으로 되어 실행 사이클이 같은 것을 볼 수 있다.



(a) Convolution (b) Edge detection

그림 5.2. Execution cycles of cacheable attribute in pLIAM (4 caches, 4 channels).

제시한 기술이 어느 특정 환경에서만이 아닌 다른 환경에서도 적용될 수 있음을 확인하기 위해 채널 수를 변경하여 실험을 진행하였다. 앞선 실험에서는 채널을 4개로 설정하였고 이를 2개로 변경하였다. 식 4.4에 만족하는 이미지의 크기에 pLIAM을 적용하였다. 이는 그림 5.3에서 확인할 수 있다. 그 결과 로테이티드 디스플레이, 로테이티드 프리뷰, 컨볼루션, 엣지 디텍션 각각 평균 15.6%, 13.1%, 13.2%, 7.4%의 감소율을 얻을 수 있었고 최대 39.75%, 32.7%, 37.5%, 13.7%의 감소율을 얻을 수 있었다. 그러나 720X480의 이미지 크기에서 T값이 식 4.4에 만족하지만 실행 사이클이 모든 실험에서 감소하지 않는 경향을 보였다. 720X480 이미지 크기의 경우 매우 작은 이미지의 크기이므로 T값에 상관없이 기존 선형 매핑 방식을 채택할 수 있으므로 문제가 되지 않는다. 이를 제외한 나머지 이미지 크기에서는 pLIAM이 적용되지 않아 실행 사이클 수가 변하지 않는 것을 확인할 수 있다. 이를 통해 어느 특정 환경에서만 적용하였을 때 효과가 있는 것이 아닌 다른 환경에서도 pLIAM이 효과가 있는 것을 확인하였다.



(a) Rotated display

(b) Rotated preview

(c) Convolution

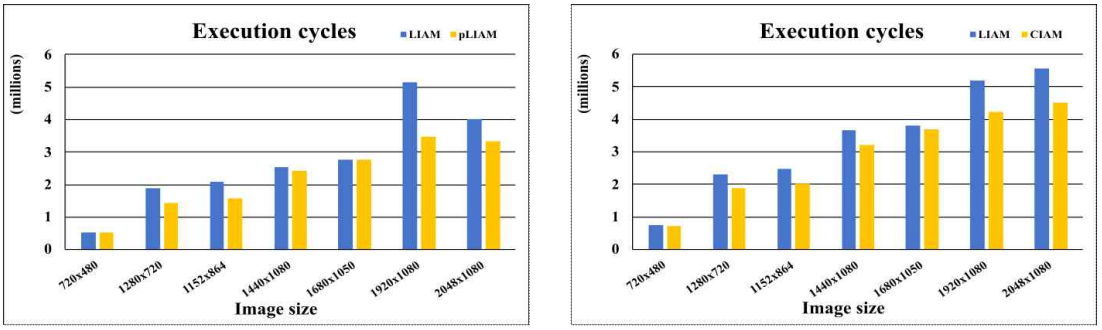
(d) Edge detection

그림 5.3. Execution cycles of pLIAM (4 caches, 2 channels).

5.2.2 CIAM의 실행 사이클

CIAM은 캐시 비트를 변환하는 방식이다. 캐시 비트를 변환하는 방식이기 때문에 논-캐시어블의 메모리 속성에서는 적용되지 않는다. 캐시어블의 메모리 속성을 가지고 있는 컨볼루션 연산과 엣지 디텍션 연산에 각각 CIAM을 적용하였다. CIAM을 적용한 결과 그림 5.4과 같은 결과를 얻을 수 있었다. CIAM의 메트릭의 경우 pLIAM의 메트릭과 동일하다. 그러므로 T값에 만족하는 이미지 크기인 1280X720, 1152X864, 1440X1080, 1920X1080, 2048X1080의 경우 CIAM을 적용한다. 그 결과 컨볼루션의 경우 평균 11.1%의 감소율, 최대 32.3%의 감소율의 결과를 얻을 수 있었고 엣지 디텍션의 경우 평균 7.4%의 감소율, 최대 14.6%의 감소율의 결과를 얻을 수 있었다. 또한 T값이 식 4.4 조건에 만족하지 않는 경우 CIAM이 적용되지 않아 실행 사이클 수가 LIAM과 CIAM이

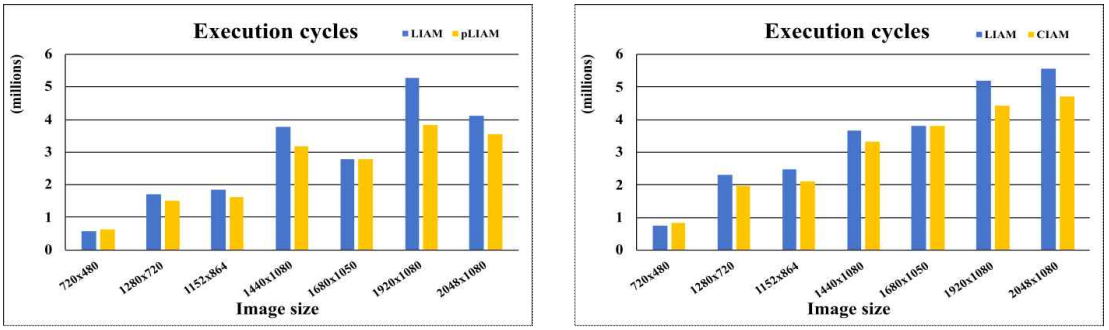
같은 것을 확인할 수 있다.



(a) Convolution (b) Edge detection

그림 5.4. Execution cycles of CIAM (4 caches, 4 channels).

CIAM이 특정 환경에서만 효과가 있는 것이 아닌 다른 환경에서도 효과가 있다는 것을 확인하기 위해 메모리 채널의 개수를 변경하여 실험을 진행하여 보았다. 기존의 모델에서의 채널 개수는 4개였다. 이를 2개를 변경하여 실험을 진행하였다. 그 결과 그림 5.5과 같이 T값에 만족하는 경우의 이미지 크기인 1280X720, 1152X864, 1440X1080, 1920X1080, 2048X1080의 경우 실행 사이클이 감소한 것을 확인할 수 있다. 그러나 720X480의 이미지 크기에서 T값이 식 4.4에 만족하지만 실행 사이클이 모든 실험에서 감소하지 않는 경향을 보였다. 720X480 이미지 크기의 경우 매우 작은 이미지의 크기이므로 T값에 상관없이 기존 선형 매핑 방식을 채택할 수 있으므로 문제가 되지 않는다. 컨볼루션의 경우 평균 13.2%, 최대 37.5%의 감소율을 확인할 수 있었고 엣지 디텍션의 경우 평균 7.8%, 최대 15.3%의 감소율을 확인할 수 있었다. 이를 통해 특정 환경에서만 제시한 기술이 효과가 있는 것이 아닌 다른 환경에서도 효과가 있다는 것을 확인할 수 있다.



(a) Convolution (b) Edge detection

그림 5.5. Execution cycles of CIAM (4 caches, 2 channels).

5.3 부하 균형 측정

실행 사이클이 줄어든다는 것이 트랜잭션들이 동시에 여러 개의 메모리에 할당되고 있다는 것을 확인하기 위해 부하 균형을 측정하였다. 이를 위해 메모리 또는 캐시에 있는 아웃스탠딩 리퀘스트의 개수를 측정하였다. 아웃스탠딩 리퀘스트는 메모리나 캐시 내에 요청 큐 (Queue)에 저장되어있는 리퀘스트 (Request)를 말한다. 이 아웃스탠딩 리퀘스트가 특정 메모리 큐에 쌓여 있고 동시에 다른 메모리 큐는 비어있다면 이는 부하 균형이 좋지 않다. 그러므로 아웃스탠딩 리퀘스트의 개수가 여러 개의 메모리에 고르게 분배된 것이 부하 균형에 좋다고 볼 수 있다. 또한 이는 인터리빙이 원활하게 잘 되고 있음을 의미한다.

5.3.1 pLIAM의 아웃스탠딩 리퀘스트 개수

아웃스탠딩 리퀘스트를 측정하기 위해 1920X1080 이미지 크기에서 측정을 진행하였다. 또한 각 메모리 혹은 캐시에서 20 사이클마다 리퀘스트 큐에 있는 트랜잭션의 개수를 측정하였다. 이때 어느 채널의 리퀘스트 큐에 있는 트랜잭션의 개수가 0인 경우 그 메모리 혹은 캐시는 유휴상태에 있다.

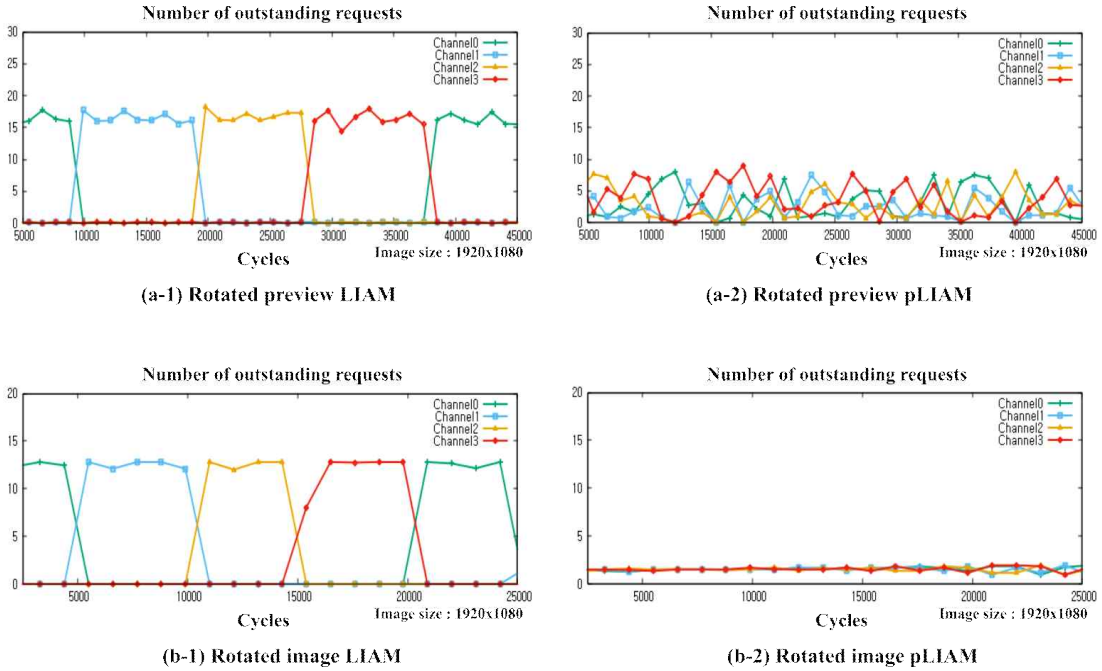


그림 5.6. Number of outstanding requests in non-cacheable attribute (pLIAM).

그림 5.6은 논-캐시어블 속성에서 pLIAM을 적용한 실험의 아웃스탠딩 리퀘스트 개수를 측정한 그래프이다. 그림 5.6 (a-1)과 (b-1)에서 아웃스탠딩 리퀘스트 수를 보았을 때 주기적으로 한 메모리에만 리퀘스트 큐에 존재하는 트랜잭션의 수가 몰리는 것을 볼 수 있고 나머지 채널에는 트랜잭션의 개수가 0인 것을 확인할 수 있다. 데이터를 처리할 때 메모리는 한 데이터를 처리하고 다음 데이터를 처리할 수 있다. 이는 여러 개의 데이터를 동시에 처리할 수 없다는 의미이다. 이 때문에 앞선 경우에 데이터를 처리할 때 지연 시간이 생긴다. pLIAM을 적용한 경우인 (a-2)와 (b-2)의 결과를 보았을 때 어느 한 채널에 트랜잭션이 몰리지 않고 동시에 여러 개의 트랜잭션이 다중의 부하들에 골고루 분배되고 있음을 확인할 수 있다. 이를 통해 pLIAM을 적용할 때 부하 균형을 높일 수 있다는 것을 알 수 있다. 또한 어느 특정 구간에서만 트랜잭션의 분배가 잘 되는 것이 아닌 패턴적으로 항상 트랜잭션의 분배가 잘 되는 것을 확인하였다.

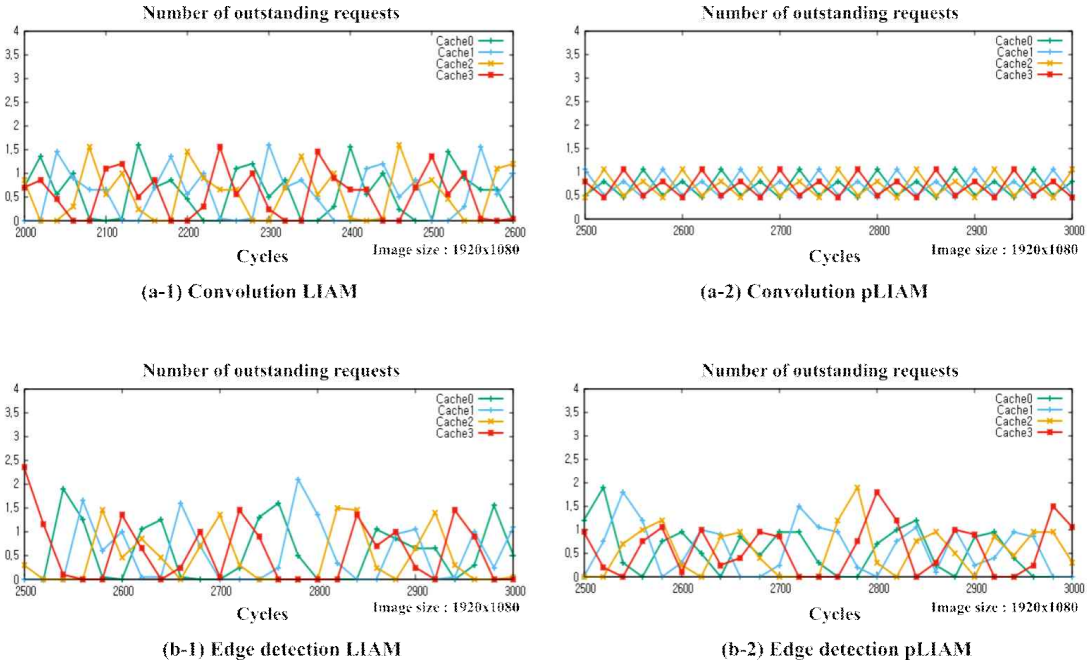


그림 5.7. Number of outstanding requests in cacheable attribute (pLIAM).

그림 5.7은 캐시어블 속성에서 pLIAM을 적용하였을 때의 아웃스탠딩 리퀘스트의 수를 그래프로 나타낸 것이다. 컨볼루션이나 엣지 디텍션에서 데이터를 처리할 때 블록 단위로 수직 처리와 선형 처리를 반복하기 때문에 그림 5.6과는 다르게 특정 주기의 사이클에 데이터가 완전하게 한 부하로만 몰리는 현상은 일어나지 않는다. 그러나 pLIAM을 적용하였을 때 그림 5.7의 (a-1), (b,1)과 (a-2), (b-2)를 비교하였을 때 유휴상태인 메모리의 개수가 줄어든 것을 확인할 수 있다. 또한 컨볼루션의 경우 pLIAM을 적용한 후 부하에 균형적으로 트랜잭션들이 할당된 것을 볼 수 있다. 이를 통해 pLIAM을 적용하였을 때 부하 균형이 향상됨을 확인하고 인터리빙이 원활하게 된다는 것을 확인할 수 있었다. 또한 캐시어블 속성의 경우에도 어느 특정 구간에서만 트랜잭션의 분배가 잘 되는 것이 아닌 다른 구간에서도 분배가 잘 되는 것을 확인하였다.

5.3.2 CIAM의 아웃스탠딩 리퀘스트 개수

CIAM을 적용하였을 때의 아웃스탠딩 리퀘스트 개수를 구하기 위해 pLIAM일 때와 마찬가지로 1920X1080의 이미지 크기에서 실험을 진행하였다. 20사이클마다의 캐시 내의 리퀘스트 큐에 존재하고 있는 트랜잭션의 개수를 측정하였다. 그림 5.8은 캐시어블 속성에 CIAM을 적용하여 아웃스탠딩 리퀘스트 개수를 그래프로 표현한 것이다. (a-1)과 (b-1)에서 컨볼루션과 엣지 디텍션 연산 처리 과정 방법에 따라 특정 사이클에 한 캐시에만 트랜잭션이 쌓여 있지는 않지만, CIAM을 적용하였을 때 (a-2)와 (b-2)에서 확인할 수 있듯이 유휴상태인 캐시가 줄어든 것을 확인할 수 있다. 또한 각 캐시의 리퀘스트 큐 내에 트랜잭션의 개수가 적절히 분배된 것을 확인할 수 있다. 이를 통해 CIAM 적용 시 부하 균형에 효과적인 것을 확인할 수 있다. 이는 인터리빙이 잘 되고 있다는 것을 의미한다. 또한 어느 특정 구간에서만 CIAM을 적용하였을 때 효과가 있는 것이 아닌 전체 구간에서도 효과가 있다는 것을 확인하였다.

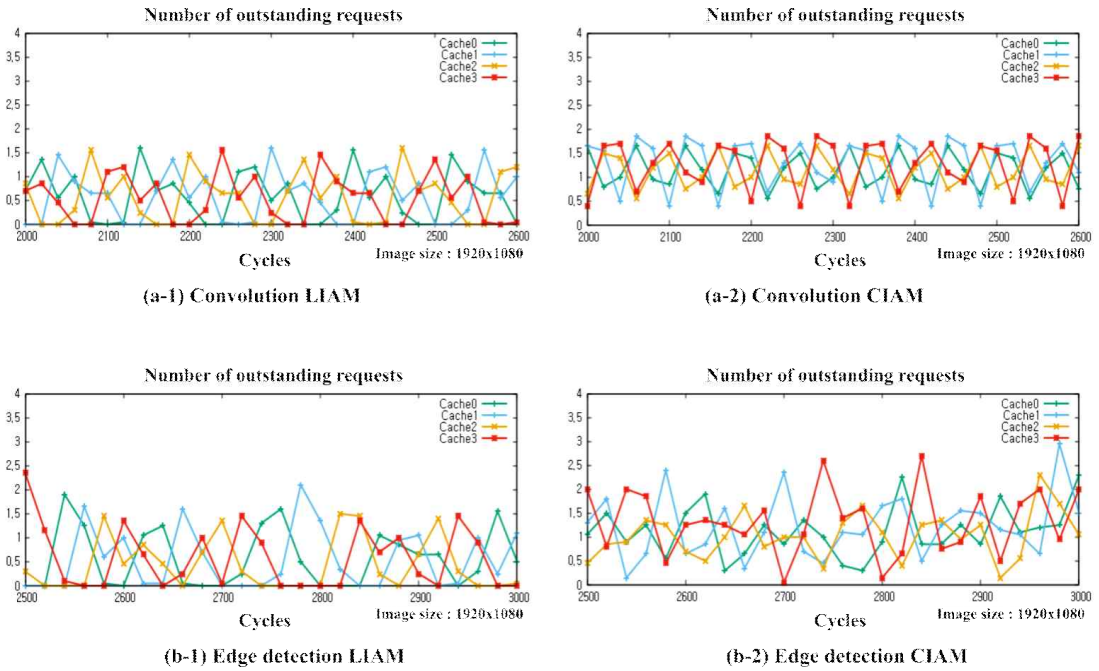


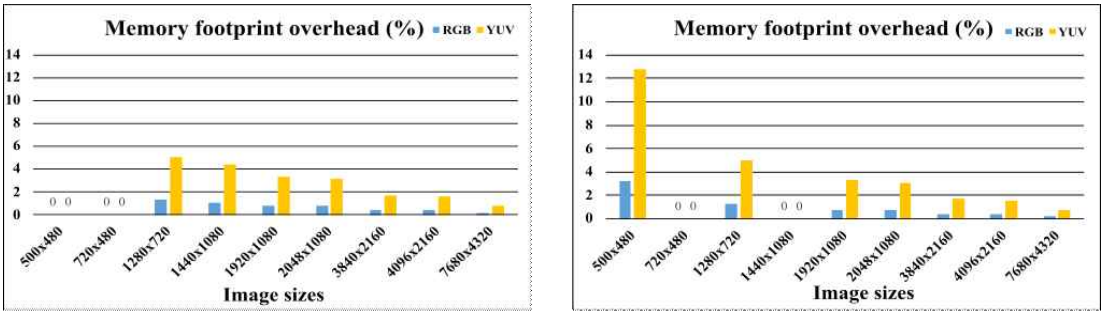
그림 5.8. Number of outstanding requests in cacheable attribute (CIAM).

5.4 오버헤드 측정

5.4.1 pLIAM의 오버헤드

pLIAM은 소프트웨어를 변경하여 기존의 방식의 문제를 해결한 기술이다. 이 기술을 구현하는 과정에서 추가의 메모리 할당이 필요하다. 이에 대한 메모리 풋 프린트 오버헤드(Memory footprint overhead)를 측정하였다. 메모리 풋 프린트는 프로그램 실행 중에 사용하거나 참조하는 메모리의 총량이다. 그림 5.9의 (a)는 메모리 채널이 4개일 경우의 메모리 추가 할당률이다. 이때 500X480, 720X480의 경우 pLIAM이 적용되지 않아 추가 메모리 할당이 필요하지 않기 때문에 메모리 추가 할당률은 0이다. 나머지 이미지 크기에서 컬러 이미지에 사용되는 RGB 포맷에서의 추가 메모리 할당률은 1% 이내로 굉장히 적은 것을 알 수 있다. 흑백 이미지에 사용되는 YUV 포맷에서의 추가 메모리 할당률은 최대 4.4%이지만 이 또한 그렇게 크지 않다. 또한 이미지 크기가 커질수록 추가 메모리 할당률은 줄어드는 것을 볼 수 있다.

그림 5.9의 (b)는 채널의 개수가 8개일 경우의 메모리 추가 할당률이다. 채널 개수가 8개일 경우 720X480, 1440X1080의 경우 pLIAM이 적용되지 않기 때문에 메모리 추가 할당이 필요하지 않기 때문에 메모리 추가 할당률은 0이다. 500X480의 경우 메모리 추가 할당률이 RGB 포맷의 경우 3.2%, YUV 포맷의 경우 12.8%로 다른 이미지 크기와는 비교하였을 때 다소 높은 경향이 있다. 그러나 500X480의 경우 이미지 크기가 작으므로 pLIAM을 적용하지 않을 수 있다. 500X480의 이미지를 제외한 나머지의 크기일 때 RGB 포맷의 경우 1% 이내로 메모리 추가 할당률이 아주 적은 것을 확인할 수 있다. 또한 YUV 포맷의 경우 최대 5%이고 이미지 크기가 커질수록 메모리 할당률은 줄어드는 것을 볼 수 있다.



(a) 4 channels (b) 8 channels

그림 5.9. Memory footprint overhead of pLIAM.

5.4.2 CIAM의 오버헤드

CIAM은 기존 주소 매핑 방식의 문제점을 해결하기 위해 하드웨어 요소를 추가한 방식이다. 기존의 하드웨어에서 새로운 하드웨어 요소가 추가되었기 때문에 이에 따른 오버헤드가 발생한다. 이를 측정하기 위해 Vivado 툴을 사용하였고 xc7s50fgga484-2 FPGA 디바이스에 합성하였다. 표 5.4는 기존 선형 방식인 LIAM을 FPGA에 합성하였을 때의 결과이고 표 5.5는 CIAM을 FPGA에 합성하였을 때의 결과이다. 이때 LUT (Look Up Table)는 룩업테이블로 연산에 대해 미리 계산된 결괏값을 가진 배열이고 룩업테이블을 사용할 때 결과를 계산하는 시간보다 더 빠르게 결괏값을 얻을 수 있다. FF (Flip Flop)는 플립플롭으로 1비트의 정보를 유지할 수 있는 논리회로이다. DSP (Digital Signal Processor)는 디지털 신호 처리 장치로 디지털 연산으로 신호를 처리하는 집적회로이다. 디지털 신호 처리 장치는 높은 수준의 병렬처리가 가능하여 빠른 속도로 기능을 수행할 수 있다. IO (In Out)는 입력/출력으로 데이터의 모든 항목의 입출력을 말한다. CIAM과 비교하기 위해 표 5.4에서 LUT 수에 주목하면 기존 LIAM을 사용할 때 32개를 사용하고 이용률은 0.1%이다. 이와 비교하여 표 5.5에 LUT의 개수는 362개로 기존 LIAM과 비교하였을 때 대략 10배 이상의 차이가 발생한다. 개수로 비교하였을 때 차이가 크다고 볼 수 있다. 그러나 사용 가능한 LUT 개수인 32,600개 중 362개가 사용되어 이용률은 1.11%이다. 그러므로 LUT 사

용량이 많지 않은 것을 알 수 있다. 이는 표 5.4와 표 5.5에서 비교해 볼 수 있다.

표 5.4. Hardware overhead in LIAM.

Resource	Utilization	Available	Utilization %
LUT	32	32600	0.10
FF	32	65200	0.05
DSP	1	120	0.83
IO	113	250	45.20

표 5.5. Hardware overhead in CIAM.

Resource	Utilization	Available	Utilization %
LUT	362	32600	1.11
FF	32	65200	0.05
DSP	1	120	0.83
IO	113	250	45.20

제 6 장 결론 및 추후 과제

6.1 결론

메모리 시스템 성능은 주소 패턴과 부하 분포에 의해 크게 영향받는다. 본 논문에서는 SoC의 메모리 계층 전반에 걸쳐 부하 균형이 향상되기 위해 이미지 크기 패딩 방식과 캐시 비트 변환 방식을 제시하였다. 이미지 패드 크기는 트래픽 주소 패턴, 이미지 크기, 하드웨어 메모리 구성 및 할당된 메모리 속성에 따라 결정된다. 본 논문에서는 메모리 활용도와 성능을 개선하고 오버헤드를 측정하였다. 제시된 연구는 부하 균형이 향상되는 기법과 메모리 인터리빙을 이용하여 메모리 활용도가 향상될 수 있다. 이미지 크기를 적응적으로 패딩 하는 법과 캐시 비트 변환 기법을 통해 메모리 트래픽이 더 나은 인터리브를 달성할 수 있다. 또한 이미지 크기가 충분히 크고 트래픽 주소 패턴이 비선형일 때 성능이 향상될 수 있다.

메모리 처리 지연 시간이 줄어드는 것을 확인하기 위한 실험을 진행할 결과 pLIAM을 적용하였을 때 논-캐시어블 속성의 경우 최대 34.8%가 감소하는 것을 확인할 수 있었고 캐시어블 속성의 경우 최대 30.1%가 감소하는 것을 확인할 수 있었다. CIAM을 적용하였을 때 최대 32.3%가 감소하는 것을 확인할 수 있었다. 또한 특정 환경에서만 효과가 있는 것이 아님을 확인하기 위해 메모리 채널의 개수를 2개로 줄여 실험을 진행하였다. 진행한 결과 pLIAM과 CIAM 각각 최대 39.8%, 27.4%로 줄어드는 것을 확인할 수 있었다. 부하 균형이 잘 이루어지고 있는지에 대한 실험에서 적용 전후의 그래프를 비교한 결과 눈에 띄게 동시에 여러 메모리에 골고루 분배되고 있음을 확인하였다.

제시한 두 기술에는 각각의 오버헤드가 발생한다. 이미지를 패딩 하는 기술에서는 추가 메모리 공간이 필요할 수 있다. 이 방식에서는 특정 메모리 할당 오버헤드가 있지만 이미지 크기가 증가하면 오버헤드가 줄어든다. 오버헤드는 향상된 성능과 비교하였을 때 감수될 수 있다. 캐시 비트를 변환하는 기술에서는 하드웨

어 오버헤드가 발생한다. 추가되는 하드웨어의 개수를 전체 하드웨어에 비교하였을 때 이는 아주 적은 숫자였다.

6.2 향후 연구

본 연구에서는 임베디드 시스템의 특수 목적 I/O (Input/Output) 장치 가속기를 위한 2D 데이터의 주소 매핑에 중점을 둔다. 따라서 본 기술의 적용은 이미지 처리와 같은 고대역폭의 2D 데이터 애플리케이션으로 제한된다. 주소 패턴을 알 수 없는 다목적 또는 일반 시스템에 대한 주소 매핑을 추가로 분석할 수 있다. 이 제안을 일반화하기 위해 트래픽 패턴을 감지하는 정교한 하드웨어 설계를 추가로 개발할 수 있다. 또한 논-캐시어블 속성에서 여러 개의 마스터를 사용하는 경우 뱅크 충돌이 일어나는 현상을 발견할 수 있었다. 뱅크 충돌도 함께 고려하는 주소 매핑 방식을 또한 분석할 수 있다.

참고문헌

[1] Zhang, Z., Zhu, Z., and Zhang, X. Breaking address mapping symmetry at multi-levels of memory hierarchy to reduce DRAM row-buffer conflicts. The Journal of Instruction-Level Parallelism, 3, pp. 29-63. (2001).

[2] Kaseridis, D., Stuecheli, J., and John, L. K. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture pp. 24-35. (2011, December).

[3] Shao, J., and Davis, B. T. The bit-reversal SDRAM address mapping. In Proceedings of the 2005 workshop on Software and compilers for embedded systems pp. 62-71. (2005, September).

[4] Wei, R., Li, C., Chen, C., Sun, G., and He, M. Memory access optimization of a neural network accelerator based on memory controller. Electronics, 10(4), 438. (2021).

[5] Wang, M., Zhang, Z., Cheng, Y., and Nepal, S. Dramdig: A knowledge-assisted tool to uncover dram address mapping. In 2020 57th ACM/IEEE Design Automation Conference (DAC) pp. 1-6. (2020, July).

[6] Zhu, Z., Cao, J., Li, X., Zhang, J., Xu, Y., and Jia, G. Impacts of memory address mapping scheme on reducing DRAM self-refresh power for mobile computing devices. IEEE Access, 6, pp. 78513-78520. (2018).

[7] Islam, M., Shaizeen, A. G. A., Jayasena, N., and Kotra, J. B.U.S. Patent No. 11,487,447. Washington, DC: U.S. Patent and Trademark Office. (2022).

[8] Shao, J., and Davis, B. T. A burst scheduling access reordering mechanism. In 2007 IEEE 13th International Symposium on High Performance Computer Architecture pp. 285-294. (2007, February).

[9] Jia, W., Shaw, K. A., and Martonosi, M. MRPB: Memory request prioritization for massively parallel processors. In 2014 IEEE 20th international symposium on high performance computer architecture (HPCA) pp. 272-283. (2014, February).

[10] Lee, Y., Kim, J., Jang, H., Yang, H., Kim, J., Jeong, J., and Lee, J. W. A fully associative, tagless DRAM cache. ACM SIGARCH computer architecture news, 43(3S), pp. 211-222. (2015).

[11] Fang, Z., Zheng, B., and Weng, C. Interleaved multi-vectorizing. Proceedings of the VLDB Endowment, 13(3), pp. 226-238. (2019).

[12] Kharbutli, M., Irwin, K., Solihin, Y., and Lee, J. Using prime numbers for cache indexing to eliminate conflict misses. In 10th International Symposium on High Performance Computer Architecture (HPCA'04) pp. 288-299. (2004, February).

[13] Ghasempour, M., Jaleel, A., Garside, J. D., and Luján, M. Dream: Dynamic re-arrangement of address mapping to improve the performance of drams. In Proceedings of the Second International Symposium on Memory Systems pp. 362-373. (2016, October).

[14] Cypher, R. E. U.S. Patent No. 7,318,114. Washington, DC: U.S. Patent and Trademark Office. (2008).

[15] Sato, M., Han, C., Komatsu, K., Egawa, R., Takizawa, H., and Kobayashi, H. An energy-efficient dynamic memory address mapping mechanism. In 2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII) pp. 1-3. (2015, April).

[16] Hur, J. Y., Rhim, S. W., Lee, B. H., and Jang, W. Adaptive linear address map for bank interleaving in DRAMs. IEEE Access, 7, pp. 129604-129616. (2019).

[17] Chavet, C., Coussy, P., Urard, P., and Martin, E. Static address generation easing: a design methodology for parallel interleaver architectures. In 2010 IEEE International Conference on Acoustics, Speech and Signal Processing pp. 1594-1597. (2010, March).

[18] Lin, H., and Wolf, W. Co-design of interleaved memory systems. In Proceedings of the eighth international workshop on Hardware/software codesign pp. 46-50. (2000, May).

[19] Khan, A., Al-Mouhamed, M., Fatayar, A., Almousa, A., Baqais, A., and Assayony, M. Padding free bank conflict resolution for cuda-based matrix transpose algorithm. In 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD) pp. 1-6. (2014, June).

[20] Xu, R., and Li, Z. A sample-based cache mapping scheme. ACM SIGPLAN Notices, 40(7), pp. 166-174. (2005).

[21] Wu, S., Wang, G., Tang, P., Chen, F., and Shi, L. Convolution with even-sized kernels and symmetric padding. Advances in Neural Information Processing Systems, pp. 32. (2019).

[22] Hashemi, M. Enlarging smaller images before inputting into convolutional neural network: zero-padding vs. interpolation. Journal of Big Data, 6(1), pp.1-13. (2019).

[23] Rivera, G., and Tseng, C. W. Data transformations for eliminating conflict misses. In Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation pp. 38-49. (1998, May).

[24] Hong, C., Bao, W., Cohen, A., Krishnamoorthy, S., Pouchet, L. N., Rastello, F., ... and Sadayappan, P. Effective padding of multidimensional arrays to avoid cache conflict misses. ACM SIGPLAN Notices, 51(6), pp. 129-144. (2016).

[25] Kazuhisa, I., Motoki, O., and Hironori, K. Cache Optimization for Coarse Grain Task Parallel Processing using Inter-Array Padding. Department of Computer Science, Waseda University. Tokyo, Japan.

[26] Vera, X., Llosa, J., and González, A. Near-optimal padding for removing conflict misses. In International Workshop on Languages and Compilers for Parallel Computing Berlin, Heidelberg: Springer Berlin Heidelberg. pp. 329-343. (2002, July).

[27] Bilgic, B., Horn, B. K., and Masaki, I. Efficient integral image computation on the GPU. In 2010 IEEE Intelligent Vehicles Symposium pp. 528-533. (2010, June).

[28] Zhang, Q., Li, Q., Dai, Y., and Kuo, C. C. Reducing memory bank conflict for embedded multimedia systems. In 2004 IEEE International Conference on Multimedia and Expo (ICME)(IEEE Cat. No. 04TH8763) Vol. 1, pp. 471-474. (2004, June).

[29] ARM Architecture Reference Manual, ARMv8-A Edition. Available online: http://www.arm.com (accessed on 20 May 2023).

Adaptive Address Mapping for Load Balancing in System on Chip

So-Yeon Kim

Major of Electronic Engineering

Faculty of Applied Energy System

The Graduated School

Jeju National University

Abstract

When processing data, traditional address maps often cause traffic congestion to on-chip memory components. Additionally, if the application's access pattern does not match the address map, memory utilization decreases. In order to reduce this traffic congestion and improve memory system performance, this paper proposes two methods to adaptively solve this problem for address mapping and hardware configuration. The first is a technique for padding the image size. This technique solves the above problem by changing only the elements of the

software. The second is a technique to switch the cache bits of the address. This technique is a technique that solves a problem by only changing hardware elements. Applying these two techniques can improve load balance across the memory structure. The design is presented and performance experiments are performed, mainly targeting high-bandwidth image processing applications. In addition, the resulting overhead is measured to compare reality from an economic perspective. As a result, experiments showed that the presented design was able to improve the load balance rate while minimizing memory space overhead.