creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학박사학위논문

DRAM 호스트 인터페이스 기반 고성능 스토리지의 설계와 구현

Design and Implementation of A High Performance Storage Leveraging the DRAM Host Interface

2016년 8월

서울대학교 대학원
전기.컴퓨터공학부
서 성 용

DRAM 호스트 인터페이스 기반 고성능 스토리지의 설계와 구현

Design and Implementation of A High Performance Storage Leveraging the DRAM Host Interface

지도교수 민 상 렬

이 논문을 공학박사 학위논문으로 제출함
2016년 4월

서울대학교 대학원
전기.컴퓨터공학부
서 성 용

서 성 용의 박사 학위논문을 인준함
2016년 6월

위 원 장 _____ (인)
부위원장 _____ (인)
위 원 _____ (인)
위 원 _____ (인)
위 원 _____ (인)

초 록

스토리지는 중앙처리장치, 메인 메모리 등과 함께 컴퓨터 시스템의 성능을 결정하는 주요 요소이다. 컴퓨터 시스템이 처리해야 할 데이터의 양이 지속적으로 증가함에 따라 이를 효과적으로 처리할 수 있는 고성능 컴퓨터 시스템이 요구되고 있다. 이러한 고성능 컴퓨터 시스템을 구성하기 위해서는 고성능 스토리지가 필수적인데, NAND 플래시 메모리를 저장 매체로 사용함에 따라 스토리지의 성능은 비약적으로 발전하면서 고성능 스토리지에 대한 요구를 충족시킬 수 있었다.

스토리지는 스토리지 인터페이스를 통해 호스트 시스템과 연결된다. 스토리지 인터페이스는 호스트 시스템이 스토리지의 성능을 최대로 활용할 수 있도록 스토리지와 함께 발전해왔다. 향후에도 스토리지의 성능 향상에 따라 스토리지 인터페이스 또한 계속해서 변경되고 발전될 것이다. 특히 NAND 플래시 메모리의 급격한 발전과 PRAM, ReRAM 등의 고성능 차세대 메모리의 등장은 스토리지 인터페이스의 발전을 더욱 가속화할 것으로 예상된다.

한편 컴퓨터 시스템에는 주로 메인 메모리로 사용되는 DRAM을 위한 인터페이스도 존재한다. 메인 메모리의 대역폭과 레이턴시는 전체 컴퓨터 시스템의 성능에 직접적인 영향을 미치므로, DRAM 인터페이스는 스토리지 인터페이스를 포함한 다른 인터페이스에 비해 항상 고성능의 대역폭과 레이턴시를 제공해왔다. 이러한 고성능 DRAM 인터페이스를 스토리지 인터페이스로 활용할 수 있다면 기존의 스토리지 인터페이스에 비해 고속의 인터페이스를 확보할 수 있으며 별도의 스토리지 인터페이스를 개발할 필요가 없다. 그러나 스토리지 인터페이스와 DRAM 인터페

이스는 사용하는 매체의 특성에 따라서 상호 호환성 없이 발전해왔기 때문에 현재의 스토리지를 그대로 DRAM 인터페이스에 연결하여 사용하는 것은 불가능하다.

이 논문에서는 고속의 DRAM 인터페이스를 스토리지 인터페이스의 물리 계층으로 활용하는 스토리지 아키텍쳐를 설계, 구현, 평가함으로써 제안된 아키텍쳐의 타당성을 증명한다. 제안된 스토리지 아키텍쳐는 기존의 DRAM 인터페이스를 스토리지 인터페이스의 물리 계층으로 활용하기 위해서 DRAM과 동일한 동작 특성을 갖는 작은 크기의 공유 메모리 버퍼를 사용한다. 이 공유 메모리 버퍼는 스토리지에 의해 제공되며 호스트 시스템의 메모리 주소 공간에 매핑된다. DRAM 프로토콜 자체로는 제안된 스토리지 장치를 동작시킬 수 없기 때문에 제안된 스토리지 인터페이스를 기반으로 동작하는 소프트웨어 수준의 신규 스토리지 프로토콜을 정의한다. 제안된 프로토콜을 기반으로 DRAM (LPDDR3) 호스트 인터페이스를 사용하는 스토리지, 호스트 시스템을 설계하고 구현한다.

최종적으로 제안한 스토리지와 호스트 시스템을 결합하여 완전한 Android 시스템을 구성함으로써 본 연구의 타당성을 검증한다. 먼저 구현된 스토리지에 대한 정량적 평가를 통해 신규 스토리지 프로토콜이 매우 낮은 오버헤드를 갖고 있으며 제안된 스토리지가 최신의 UFS 2.0 스토리지와 비교될만한 성능을 확보했음을 보인다. 또한 제안된 스토리지에 대한 정성적 평가를 통해 해당 스토리지를 효과적으로 활용하기 위해 필요한 개선점에 대해 살펴본다.

주요어 : 모바일 스토리지, DRAM 인터페이스 스토리지, NVDIMM

학번 : 2004-23577

목 차

목차 . . . . . **vi**

그림 목차 . . . . . **viii**

표 목차 . . . . . **ix**

I. 서론 **1**

1.1 연구 동기 1

1.2 연구 내용 4

1.3 논문의 구성 7

II. 배경 지식 및 관련 연구 **10**

2.1 스토리지 인터페이스 10

2.1.1 데스크탑/서버 스토리지 인터페이스 11

2.1.2 모바일 스토리지 인터페이스 16

2.2 DRAM 19

2.2.1 DRAM 구조와 특성 20

2.2.2 DRAM 오퍼레이션 21

2.2.3 DRAM DIMM 21

2.3 NAND 플래시 메모리 24

2.3.1 NAND 플래시 메모리의 구조 24

2.3.2 NAND 플래시 메모리 오퍼레이션과 인터페이스 . 26

2.3.3 플래시 변환 계층 30

2.3.4 플래시 파일 시스템 35
2.4 NVDIMM 장치 37
2.4.1 NVDIMM-N 38
2.4.2 NVDIMM-F 41
2.4.3 NVDIMM-P 47
2.4.4 Intel DIMM 47
2.5 NVDIMM 소프트웨어 48
2.5.1 BIOS 49
2.5.2 Linux 지원 51
2.5.3 Microsoft Windows 지원 52
2.5.4 NVM Programming Model 52

III. 스토리지 인터페이스 설계 56
3.1 물리 계층 56
3.2 스토리지 프로토콜 57
3.3 고려사항 60
3.3.1 명령 실행 완료 전달 60
3.3.2 명령 제출시 무결성 보장 62
3.3.3 데이터 전송 부하 분배 63

IV. 스토리지 설계 65
4.1 컨트롤러 65
4.1.1 호스트 인터페이스 블록 66
4.1.2 백본 (backbone) 블록 68
4.1.3 플래시 메모리 인터페이스 블록 69
4.2 펌웨어 (Firmware) 70

4.2.1 호스트 인터페이스 계층 (Host Interface Layer) . . . 70
4.2.2 플래시 변환 계층 (Flash Translation Layer) 72
4.2.3 플래시 인터페이스 계층 (Flash Interface Layer) . . 73

V. 호스트 시스템 설계 75
5.1 호스트 시스템 플랫폼 하드웨어 75
5.2 부트 로더 (boot loader) 76
5.3 디바이스 드라이버 77
5.4 소프트웨어 최적화 80
5.4.1 DMA를 이용한 데이터 전송 80
5.4.2 I/O 분할 처리 81
5.4.3 캐시 무효화 비용 최소화 83
5.4.4 데이터 버퍼의 외부 단편화 최소화 83
5.4.5 Linux I/O 스케쥴러 미사용 84

VI. 평가 86
6.1 평가 환경 86
6.2 정량적 평가 88
6.2.1 프로토콜 오버헤드 88
6.2.2 스토리지 인터페이스 버퍼 크기와 최대 전송 속도 . 91
6.2.3 읽기/쓰기 성능 94
6.2.4 성능 비교 95
6.3 정성적 평가 98
6.3.1 DRAM 버스 대역폭 사용량 증가 98
6.3.2 유연한 DRAM 주소 매핑 98
6.3.3 DRAM 인터페이스와의 물리적 접속 101

6.3.4 메모리 트랜잭션 순서 보장 103
6.3.5 부팅 지원 104
6.3.6 고성능 범용 DMA 지원 107

VII. 결론 및 향후 연구 108
7.1 결론 108
7.2 향후 연구 108

참고 문헌 111

Abstract 117

그 림 목 차

그림 1. SAS 아키텍쳐 12
그림 2. NVMe 아키텍쳐 14
그림 3. UFS 아키텍쳐 18
그림 4. DRAM 셀 구조 19
그림 5. DRAMM DIMM 분류 22
그림 6. NAND 플래시 메모리 셀 어레이 구조 25
그림 7. NAND 플래시 메모리 인터페이스 29
그림 8. OneNAND 아키텍쳐 29
그림 9. 플래시 변환 계층의 위치 34
그림 10. NVDIMM-N 아키텍쳐 38
그림 11. ULLTraDIMM의 시스템 아키텍쳐 43
그림 12. 3D Xpoint가 반영된 메모리 계층 구조 47
그림 13. NVDIMM 소프트웨어 아키텍쳐 48
그림 14. NVDIMM firmware interface table 49
그림 15. 전체 시스템 아키텍쳐 56
그림 16. 스토리지 프로토콜 실행 순서 59
그림 17. 스토리지 컨트롤러 설계 66
그림 18. 펌웨어 스택 71
그림 19. 호스트 플랫폼 보드에서 DRAM 및 스토리지의 연결 . 76
그림 20. Linux 스토리지 소프트웨어 스택 78
그림 21. bio 구조체 79
그림 22. I/O 분할 처리 예제 82

그림 23. Scatter-gather IO 84
그림 24. 인터페이스 대역폭과 프로토콜 대역폭 89
그림 25. 프로토콜 처리를 위한 오버헤드 레이턴시 90
그림 26. 버퍼 용량과 요청 크기의 변동에 따른 최대 대역폭 (QD4) 92
그림 27. 버퍼 용량 부족으로 인한 성능 저하 93
그림 28. 다양한 요청 크기에서의 읽기/쓰기 성능 95
그림 29. UFS 스토리지와의 성능 비교 96

표 목 차

표 1. SAS 버전별 최대 전송속도 14
표 2. eMMC 버전별 최대 전송속도 17
표 3. 프로토타입 시스템 요약 86
표 4. 컨트롤러 칩 사양 87
표 5. 본 연구와 UFS 플랫폼의 사양 비교 95

제 1 장

서론

1.1 연구 동기

컴퓨터 시스템의 성능은 최초의 컴퓨터 시스템이 등장한 이후부터 현재에 이르기까지 계속해서 향상되어 왔다. 이러한 성능 향상은 컴퓨터 시스템을 구성하는 중앙처리장치, 메인 메모리, 스토리지 등 각각의 요소들의 성능이 개선됨에 따라 가능했다. 중앙처리장치 (CPU, Central Processing Unit)는 컴퓨터 시스템에서 연산을 담당한다. 중앙처리장치의 성능은 반도체의 집적도가 18개월마다 2배로 증가한다는 무어의 법칙 (Moore's law)에 따라 급격히 향상되어 최신 제품의 동작 속도는 수 GHz (Giga Hertz) 에 이른다.

메인 메모리는 중앙처리장치가 실행중인 프로그램과 처리중인 데이터를 저장한다. 메인 메모리로는 주로 DRAM (Dynamic Random Access Memory)을 사용하는데, ① 바이트 (byte) 단위의 접근이 가능하며, ② SRAM (Static Random Access memory) 에 비해 같은 비용으로 큰 용량을 사용할 수 있고, ③ 비교적 빠른 속도를 제공하기 때문이다. DRAM 역시 무어의 법칙에 따라 빠른 속도로 발전했으며, 한동안은 황의 법칙 (Hwang's law)에 따라 집적도가 12개월마다 2배로 증가하기도 했다.

한편 스토리지는 전원이 공급되지 않아도 프로그램과 데이터를 비휘발성으로 저장하는 역할을 담당한다. 기존에는 주로 하드디스크 (HDD, Hard Disk Drive)가 사용되었는데, 반도체인 중앙처리장치나 메인 메모

리에 비해 속도 향상이 지체되었다. 속도 향상의 지체는 반도체 소자인 DRAM과 달리 플래터 (platter), 스핀들 모터 (spindle motor), 암 (arm), 그리고 헤드 (head) 등의 기계적인 요소로 구성되는 하드디스크의 구조적인 특성에서 기인한다. 하드디스크의 정보에 대해 읽기 (read) 또는 쓰기 (write) 작업을 수행하기 위해서는 스핀들 모터와 암을 이용하여 헤드를 플래터상의 목표 위치까지 이동시켜야 한다. 이로 인해 하드디스크에서 한번의 I/O 연산을 수행하기 위해서는 헤드를 플래터상의 목표 위치까지 이동하는 시간인 탐색 시간 (seek time)이 발생하며, 이 시간은 일반적으로 수 *ms* (microsecond) 수준이다. 따라서 하드디스크는 연속된 데이터에 대한 순차 접근 (sequential access) 성능은 비교적 우수하나, 임의의 데이터에 대한 임의 접근 (random access) 성능은 매우 떨어지는 단점이 있다.

하지만 NAND 플래시 메모리 (NAND flash memory)가 등장하고, 이를 스토리지의 저장 매체로 사용하면서 스토리지의 성능은 급격히 발전했다. NAND 플래시 메모리는 반도체 소자이므로 기존의 하드디스크와 달리 기계적 요소가 없이 전기적 신호로만 동작하므로 NAND 플래시 메모리로 구성된 스토리지는 임의 접근 성능이 뛰어나다. 또한 크기가 작기 때문에 하나의 스토리지에 다수의 NAND 플래시 메모리 칩을 장착할 수 있고, 다수의 NAND 플래시 메모리를 동시에 동작시켜 병렬성을 높임으로써 높은 성능을 제공할 수 있다. 이러한 NAND 플래시 메모리의 특성을 이용하여 스토리지의 성능은 기존에 비해 급격히 빠른 속도로 발전했다.

호스트와 각 장치를 연결하는 호스트 인터페이스 (host interface) 또한 계속 발전하면서 DRAM 과 스토리지의 향상된 성능을 효과적으로 활용할 수 있었다. DRAM 인터페이스의 최신 버전인 DDR4는 최대 25.6 GB/s 의 전송속도를 지원한다. 스토리지 인터페이스는 SATA (Serial ATA), SAS (Serial Attached SCSI), NVMe (NVM express), eMMC (Embedded

MultiMedia Card), UFS (Universal Flash Storage) 등이 있다. 데스크탑 환경에서는 주로 SATA (Serial ATA) 인터페이스를 사용하는데, 최신 버전은 SATA3 이며 최대 전송속도가 600 MB/s 이다. 고성능을 필요로 하는 엔터프라이즈 환경에서는 SAS , NVMe 등의 인터페이스를 사용한다. 이들의 최신 버전은 각각 4.8 GB/s (2 레인), 8 GB/s (PCIe 3 세대 4 레인) 의 전송속도를 지원한다. 특히 NVMe 는 하드디스크가 아닌 NAND 플래시 메모리와 같은 반도체 소자로 구성된 솔리드 스테이트 디스크 (SSD, Solid State Disk)를 위해 제안된 새로운 인터페이스이다. 앞으로도 호스트 인터페이스는 각 장치의 성능 향상에 따라 계속해서 변경되고 발전할 것으로 예상된다.

그런데 DRAM과 스토리지는 데이터를 저장한다는 공통점에도 불구하고 서로 다른 인터페이스를 사용해왔다. 그 이유는 I/O 연산의 단위와 응답시간 (latency)의 차이에서 기인한다. 첫째, I/O 연산의 단위가 DRAM 은 바이트 또는 캐시라인 (cacheline, 주로 64 바이트)이지만, 스토리지는 섹터 (sector, 주로 512 바이트)이다. 둘째, DRAM 은 수 *ns* 단위의 고정된 응답시간을 갖는다. 그러나 스토리지는 타임아웃 (timeout) 이외에는 고정된 응답시간을 보장하지 않으며, I/O 연산이 완료되면 호스트에게 통보하는 방식으로 동작한다. 이와 같은 본질적인 특성 차이로 인해 DRAM 과 스토리지 인터페이스는 앞으로도 일정 기간 현재이 독자적으로 발전할 것으로 예상된다.

한편 DRAM과 스토리지 인터페이스를 통합하고자 하는 연구도 진행되고 있다. 이는 상변화 메모리 (PRAM, Phase change Random Access Memory), 자기저항 메모리 (MRAM, Magnetoresistive Random Access Memory) 등의 고성능 차세대 메모리의 등장과 NAND 플래시 메모리의 성능 향상으로 인해 DRAM 과의 성능 격차가 줄어들어고 있기 때문이다.

DRAM 인터페이스를 활용하면 매우 높은 대역폭과 낮은 응답시간을 갖는 인터페이스를 확보할 수 있다. 하지만 DRAM 인터페이스의 특성 (① I/O 연산 단위, ② 고정된 응답시간) 때문에 기존의 DRAM 인터페이스에 DRAM 이외의 메모리 소자나 장치를 연결해서 바로 사용할 수는 없다.

본 논문에서는 이러한 문제를 해결하고자 DRAM 인터페이스를 물리적인 변경 없이 스토리지 인터페이스로 사용하는 방법과 이 인터페이스를 사용하는 스토리지 및 호스트 시스템에 대해 연구한다. 먼저 DRAM 인터페이스를 변경하지 않고 스토리지 인터페이스로 사용하기 위한 시스템 아키텍쳐 및 DRAM 인터페이스 위에서 동작하는 프로토콜을 제안한다. 제안된 아키텍쳐와 프로토콜을 기반으로 스토리지와 호스트 시스템을 설계하고 구현한다. 그리고 구현된 스토리지와 호스트 시스템을 통합하여 해당 스토리지로 구동되는 컴퓨터 시스템을 구축하고 평가함으로써 본 논문에서 제안한 기법의 타당성을 증명한다. 마지막으로 결론과 향후 연구 방향을 제시한다.

1.2 연구 내용

본 논문에서는 DRAM 인터페이스를 호스트 인터페이스로 사용하는 스토리지와 해당 스토리지를 사용하는 호스트 시스템에 대해 연구한다 [1]. 먼저 해당 스토리지를 사용하는 전체 시스템 아키텍쳐를 정의하고, 호스트 시스템과 제안된 스토리지가 연결되고 자료를 교환하는 방식을 기술한다. 해당 스토리지는 기존의 시스템과 다르게 별도의 호스트 인터페이스를 위한 인터페이스가 아닌 DRAM 컨트롤러에 DRAM 인터페이스를 통해 연결된다. 여기서 스토리지는 호스트 시스템과의 통신을 위해서 호스트 시스템과 스토리지 내부에서 모두 접근 가능한 작은 크기의 공유

메모리 버퍼를 제공한다. 호스트의 DRAM 컨트롤러는 스토리지의 공유 메모리 버퍼를 일반적인 DRAM 장치와 동일하게 인식하며 같은 방식으로 제어한다.

제안된 물리 계층 수준의 연결을 기반으로 호스트 시스템과 스토리지가 공유 메모리 버퍼에서 사용하는 소프트웨어 수준에서 정의된 프로토콜을 제안한다. DRAM 인터페이스에서 기본으로 사용하는 프로토콜만으로는 스토리지에서 사용하는 섹터 수준의 읽기 및 쓰기 연산을 수행할 수 없으므로 소프트웨어 수준에서 정의된 신규 프로토콜이 필요하다. 본 논문에서 제안한 프로토콜은 공유 메모리 버퍼를 ① 제출 큐 (SQ, Submission Queue), ② 완료 큐 (CQ, Completion Queue), ③ 데이터 버퍼 (DB, Data Buffer)로 나누어 사용한다. 호스트 시스템은 제출 큐에 스토리지 명령을 기록하고, 스토리지 장치는 제출 큐에서 처리할 명령을 가져와서 처리한 후, 완료 큐에 결과를 기록한다. 읽기 또는 쓰기 데이터는 데이터 버퍼를 통해서 교환된다.

제안된 물리 계층 수준의 연결 및 프로토콜을 사용하는 아키텍쳐에 대한 최적화를 위해 필요한 기법을 연구한다. 기존의 호스트 시스템은 호스트 시스템과 스토리지 장치간의 명령과 자료 이동을 전용 컨트롤러가 담당했으나, 본 논문에서 제안한 방식에서는 호스트의 중앙처리장치가 이 역할을 담당하게 된다. 따라서 중앙처리장치의 부하를 줄이고 성능을 개선하기 위해 중앙처리장치에 내장된 DMA 컨트롤러를 활용하는 방법, GPIO를 활용하는 방법 등을 살펴본다.

전체 시스템 아키텍쳐 설계가 완료되면 이를 위한 NAND 플래시 메모리 기반 스토리지를 설계한다. 스토리지는 크게 스토리지 컨트롤러 (controller)와 펌웨어 (firmware)로 구분된다. 스토리지 컨트롤러는 호스트 인터페이스, 펌웨어 실행을 위한 기반 장치 (중앙처리장치, SRAM 등),

NAND 플래시 메모리 컨트롤러 등으로 구성된다. 스토리지 컨트롤러가 제공하는 호스트 인터페이스는 공유 메모리 버퍼를 제공하며, 이것은 단일 포트 (single port) SRAM 을 이용해 외부의 호스트 시스템, 내부의 스토리지 컨트롤러 모두에서 접근이 가능하도록 설계되었다. 펌웨어는 스토리지 컨트롤러가 제공하는 중앙처리장치와 SRAM을 이용해 실행되며, 호스트 인터페이스 계층 (host interface layer), 플래시 메모리 변환 계층 (flash memory translation layer), 플래시 메모리 인터페이스 계층 (flash memory interface layer) 로 구성된다.

또한 제안된 스토리지를 사용할 수 있는 호스트 시스템 하드웨어와 소프트웨어를 설계한다. 호스트 시스템 하드웨어는 모바일 플랫폼을 위한 개발보드를 기반으로 하며, 변경은 제안한 스토리지를 호스트 DRAM 컨트롤러에 연결하기 위한 PCB 수정에 국한된다. 기존의 DRAM 인터페이스를 물리 계층을 연결하기 위해 그대로 사용하므로 DRAM 컨트롤러에 대한 변경은 불필요하다. 호스트 시스템 소프트웨어는 부트로더 (bootloader)와 운영체제 (operating system)의 디바이스 드라이버 (device driver)가 필요하다. 부트로더는 제안된 스토리지의 공유 메모리 버퍼를 호스트 시스템의 물리 주소 공간에 사상 (map) 하는 역할을 담당하며, 디바이스 드라이버는 해당 스토리지를 운영체제에 블록 장치로 등록하고 스토리지를 제어한다.

위의 설계를 바탕으로 구현을 완료하고 본 연구에서 제안한 기법의 타당성을 검증하고 성능을 정량적, 정성적으로 평가한다. 먼저 제안된 프로토콜의 오버헤드를 평가하고 UFS 2.0 프로토콜과 비교함으로써 제안된 프로토콜의 타당성을 검증한다. 그 후 스토리지 인터페이스 버퍼 용량과 인터페이스 최대 성능간의 민감도를 분석하고 목표 성능을 위한 버퍼 용량에 대해 검토한다. 마지막으로 기본적인 읽기/쓰기 성능을 평가하고,

이를 최신 UFS 2.0 스토리지와 비교함으로써 제안된 스토리지에 대한 정량적 평가를 마친다. 그리고 정성적 평가를 수행함으로써 제안된 스토리지를 더욱 효과적으로 활용하기 위한 방안을 검토한다.

본 연구의 의의는 다음과 같다.

1. DRAM 인터페이스를 스토리지 인터페이스로 사용하는 스토리지 및 시스템 아키텍쳐가 가능함을 보인다.

2. 스토리지와 호스트 시스템을 실제로 구현하고 평가한다.

3. 제안된 스토리지 아키텍쳐를 현재의 호스트 시스템 환경에 적용할 때 발생하는 문제를 지적하고, 다음 세대의 시스템 아키텍쳐에서 개선할 점을 제안한다.

1.3 논문의 구성

본 논문의 구성은 다음과 같다. 제 2장에서는 본 연구의 배경 지식과 관련 연구에 대해 살펴본다. 먼저 다양한 스토리지 인터페이스 (SATA, SAS, NVMe, eMMC, UFS)에 대해 각 인터페이스의 특징과 발전 동향을 살펴본다. 그리고 DRAM의 구조 및 인터페이스 등에 대해 살펴본다. DRAM 셀 구조, 뱅크, 랭크, 채널 등의 개념과 동작 방식에 대해 살펴보고 DRAM 인터페이스와 스토리지 인터페이스의 차이를 정리한다. 또한 다양한 DRAM DIMM에 대해서도 살펴본다. 본 논문에서 제안한 스토리지는 NAND 플래시 메모리에 기반을 두고 있으므로 NAND 플래시 메모리 및 관련 기술에 대해서도 살펴본다. 마지막으로 NVDIMM 에 대해 살펴본다.

제 3장에서는 스토리지 인터페이스를 설계한다. 먼저 물리 계층 수준

에서의 연결 방식을 제안한다. 제안되는 스토리지 인터페이스는 DRAM 버스를 통해서 호스트 시스템과 연결되며, 따라서 스토리지 인터페이스는 DRAM과 같은 동작 특성을 제공해야 한다. DRAM 프로토콜을 통해 직접 스토리지 명령을 처리할 수 없기 때문에 해당 물리 계층 위에서 소프트웨어 수준의 스토리지 프로토콜을 정의하고 동작을 살펴본다. 마지막으로 스토리지 프로토콜의 설계 과정에서 고려할 점을 논의한다.

제 4장에서는 스토리지를 설계한다. 먼저 스토리지 컨트롤러에 대해 살펴본다. 제안된 스토리지 컨트롤러는 크게 호스트 인터페이스, 백본, 플래시 메모리 인터페이스의 3블록으로 구성된다. 가장 특징적인 부분은 싱글 포트 SRAM을 활용해서 호스트 시스템에 DRAM과 같은 특성을 갖는 호스트 인터페이스를 제공하는 호스트 인터페이스 블록이다. 백본 블록은 펌웨어 실행에 필요한 CPU 자원과 메모리를 제공하며, 플래시 메모리 인터페이스는 플래시 메모리를 제어하기 위한 하드웨어 장치를 제공한다. 스토리지 펌웨어는 해당 스토리지 컨트롤러 위에서 동작하며, 크게 호스트 인터페이스 계층, 플래시 변환 계층, 플래시 메모리 인터페이스 계층으로 구분된다. 호스트 인터페이스 계층은 제안된 스토리지 프로토콜을 위해 새로 설계되었으며, 스토리지 프로토콜과 스토리지 내부 요청 사이의 변환을 담당한다.

제 5장에서는 제안된 스토리지 장치를 위한 호스트 시스템을 설계한다. 먼저 호스트 시스템 하드웨어의 설계를 살펴본다. 호스트 시스템은 기존의 모바일 시스템 플랫폼 레퍼런스 보드를 일부 수정하도도록 설계되었으며, DRAM 채널 0에는 실제 DRAM이, 채널 1에는 제안된 스토리지가 연결된다. 그 후에는 호스트 시스템 소프트웨어를 살펴본다. 먼저 시스템 초기화시에 제안된 스토리지에 대한 설정을 수행할 부트 로더를 설계한다. 부트 로더는 제안된 스토리지 인터페이스에 대한 동작 타이밍 파라

미터, 주소 공간에 대한 매핑 등을 담당한다. 마지막으로 Linux 디바이스 드라이버에 대한 구현을 살펴본다. 디바이스 드라이버는 초기화 과정에서 스토리지 장치에 대한 인식, 설정, 블록 장치 등록 등을 수행한다. 그 후 상위 계층의 I/O 요청을 받아 스토리지의 제출 큐에 요청을 제출하고, 스토리지의 완료 큐로부터 요청이 완료됨을 확인하는 역할을 담당한다. 여기서 성능 향상 및 무결성 보장을 위한 몇 가지 최적화 기법을 살펴본다.

제 6장에서는 구현된 시스템을 평가한다. 여기서는 제안된 스토리지와 이를 위한 호스트 시스템을 통합하여 정량적, 정성적 평가를 수행한다. 먼저 평가 환경을 기술하는데, 프로토타입 시스템의 사양, 컨트롤러 칩의 사양, 평가 방법 등을 살펴본다. 그 후 제안된 스토리지 프로토콜의 오버헤드에 대한 평가, 스토리지 인터페이스를 위한 버퍼 크기에 대한 성능 민감도 분석, 읽기/쓰기 성능 평가, 최신 UFS 2.0 스토리지와의 성능 비교를 통해 제안된 스토리지의 타당성을 정성적으로 평가한다. 정량적 평가 후에는 정성적인 평가를 실시함으로써 제안된 스토리지를 더욱 효과적으로 활용하기 위한 방안을 검토한다.

마지막으로 제 7장에서는 연구 결과를 요약하고 향후 연구 과제를 제시한다.

제 2 장

배경 지식 및 관련 연구

이 장에서는 본 연구의 배경 지식과 관련 연구를 살펴본다. 먼저 기존의 다양한 스토리지 인터페이스와 그 특징을 확인한다. 그리고 본 연구에서 스토리지 인터페이스의 물리 계층으로 활용하고자 하는 DRAM 인터페이스에 대해 살펴본다. 한편 본 연구에서 제안한 스토리지는 NAND 플래시 메모리를 저장 매체로 사용하므로 NAND 플래시 메모리의 특징과 NAND 플래시 메모리를 위한 소프트웨어에 대해 알아본다. 마지막으로 최근 확산되고 있는 DRAM과 NAND 플래시 메모리를 기반으로 하고 있는 NVDIMM 기술과 이를 위한 소프트웨어에 대해 살펴봄으로써 이 장을 마무리한다.

2.1 스토리지 인터페이스

전통적인 컴퓨터 시스템에서는 스토리지를 호스트 시스템과 연결하기 위해 별도의 독립적인 스토리지 인터페이스를 사용한다. 호스트 시스템과 스토리지는 각각 해당 인터페이스를 위한 컨트롤러를 구비하고 있으며, 각 컨트롤러는 전용 버스 (bus)를 통해서 연결된다.

스토리지 인터페이스의 주 연산은 읽기 (read)와 쓰기 (write) 이며, 연산의 단위는 일반적으로 섹터 (sector) 를 사용한다. 스토리지에서는 대용량의 자료에 대한 입출력이 발생하므로 연산의 기본 단위인 섹터의 크기는 일반적으로 512 바이트 또는 4096 바이트를 사용한다.

스토리지 인터페이스는 환경의 특성에 따라 데스크탑/서버, 모바일 환경으로 각각 발전해왔다. 데스크탑 환경에서는 ATA, 서버 환경에서는 SCSI, 모바일 환경에서는 eMMC 가 주로 사용되고 있다. 최근에 새로 등장한 인터페이스는 데스크탑/서버 환경을 위한 NVMe, 모바일 환경을 위한 UFS 가 있다. 특히 NVMe 의 경우는 PCIe 버스 환경에서 동작하기 때문에, 모바일에서도 PCIe 환경을 구축하면 사용이 가능하다.

2.1.1 데스크탑/서버 스토리지 인터페이스

2.1.1.1 SATA (Serial ATA)

SATA (Serial ATA)는 일반적인 데스크탑 환경에서 널리 사용되는 스토리지 장치 인터페이스이며, 2003년에 표준화되었다. 기존의 PATA (Parallel ATA)는 40 또는 80선의 리본 케이블을 이용했는데, 이로 인해 데이터 버스의 동작 속도를 고속으로 높이기가 어려웠다. SATA 인터페이스는 2쌍의 전도체로 구성된 고속의 직렬 케이블을 이용함으로써 속도를 개선하였다. PATA 의 경우 UDMA 6 모드에서 최대 133 MB/s 의 전송속도를 지원했으나, SATA 는 1.0 버전의 150 MB/s 로부터 3.0 버전의 600 MB/s 까지 향상되었다.

SATA 2.0부터 본격적으로 NCQ (Native Command Queueing) 기능이 사용되기 시작했다. NCQ는 스토리지 장치가 실행해야 할 호스트 명령을 큐 (Queue)에 쌓아두고, 스토리지 장치가 명령의 선택 순서를 선택하여 실행하는 것이다. 이를 통해 전체 시스템의 성능을 개선할 수 있다. 기존의 PATA 에서도 NCQ와 유사한 TCQ (Tagged Command Queueing) 기능이 제공되었으나, 일반적으로 널리 사용되지는 않았다.

SATA 인터페이스는 HDD와 SSD (Solid State Disk)가 모두 사용하고

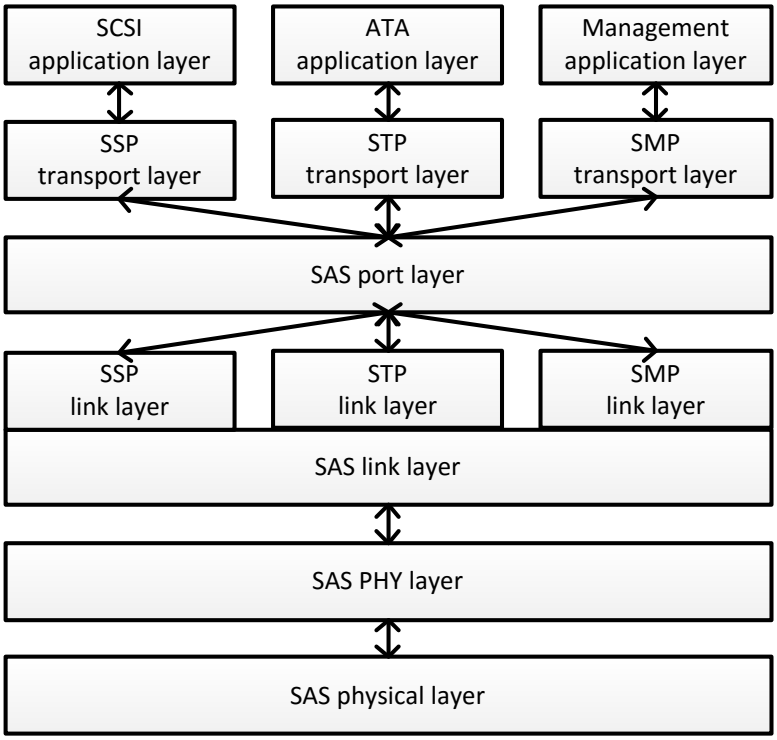

그림 1: SAS 아키텍쳐

있으나, HDD는 SATA 2.0 버전의 최대 속도인 300 MB/s 조차도 충분히 활용하지 못하는 성능을 보여주고 있다. 반면 SSD는 NAND 플래시 메모리 자체의 성능 향상과 다수의 NAND 플래시 메모리 칩을 활용한 병렬성을 통해 성능이 급격히 향상되어 SATA 3.0 버전의 최대 속도인 600 MB/s 를 뛰어넘어 더 빠른 속도를 제공하는 NVMe 등의 인터페이스를 채택하는 추세이다.

2.1.1.2 SAS (Serial Attached SCSI)

SAS (Serial Attached SCSI)는 높은 성능과 신뢰성을 제공하므로 서버 시스템에서 주로 사용된다. 기존에는 SCSI 인터페이스 또한 ATA 인터페이스와 마찬가지로 패러랠 방식을 사용했으나 속도 향상을 위해 시리얼 방식으로 변경하였다. SAS의 계층 구조는 그림 1에서 보이는 것과 같다[2].

- Initiator

  타겟 장치에 작업을 요청하고 결과를 수신하는 장치로 호스트 컨트롤러가 포함된다.

- Target

  작업을 요청받아 처리하고 결과를 송신하는 장치로 디스크나 디스크 어레이 등이 포함됨된다.

- Service delivery subsystem

  Initiator와 target 간의 정보를 전송하는 장치로 케이블 등이 포함된다.

- Expander

  다수의 SAS 최종 장치가 단일 initiator 포트에 연결되도록 하는 장치이다.

SAS의 버전별 최대 전송속도는 표 1에 정리되어 있다.

Version	Throughput (Gbit/s)
SAS-1	3.0
SAS-2	6.0
SAS-3	12.0
SAS-4	22.5

표 1: SAS 버전별 최대 전송속도

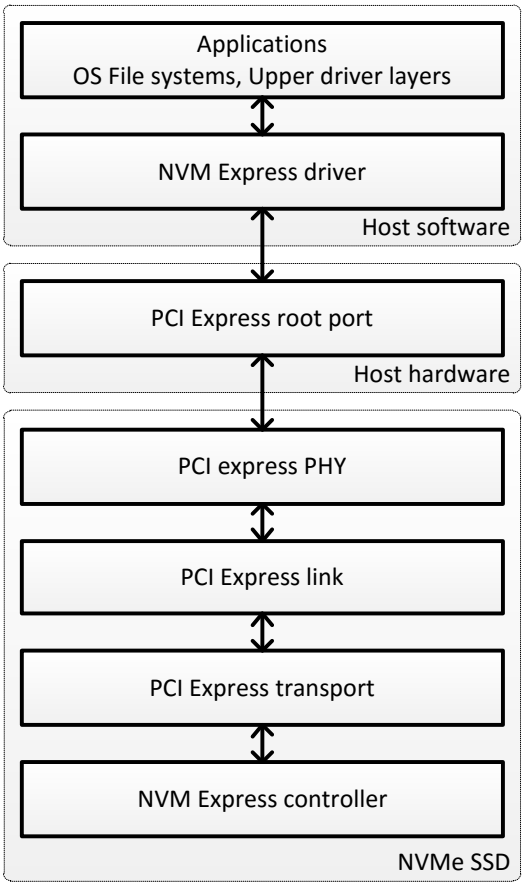

그림 2: NVMe 아키텍쳐

2.1.1.3 NVMe

NVMe (NVM express) 인터페이스는 2011년에 제안되어 사용되고 있으며 최신 버전은 1.2 이다 [3]. HDD 를 기반으로 발전한 ATA, SCSI 와 달리 처음부터 NAND 플래시 메모리와 같은 NVM (Non-Volatile Memory) 를 위해 설계되었다.

NVMe는 PCIe (PCI express)를 기반으로 하고 있으며 아키텍쳐는 그림 2와 같다. 그림 2에서 볼 수 있듯이 NVMe의 가장 큰 특징은 호스트 시스템에 해당 인터페이스를 위한 별도의 컨트롤러가 존재하지 않는다는 것이다. 대신 호스트 시스템 스토리지 인터페이스 컨트롤러의 역할은 디바이스 드라이버가 PCIe의 기능을 활용해서 수행한다. 명령 완료에 대한 인터럽트도 PCIe의 MSI-X를 이용한다.

NVMe에서 호스트 시스템과 NVMe 스토리지간의 통신은 호스트 시스템의 메모리에 큐 (queue)를 생성하고, 이를 통해서 이루어진다. 이러한 방식의 통신이 가능한 것은 PCIe 버스에 연결된 장치가 호스트 시스템의 메모리에 접근이 가능하기 때문이다.

큐는 관리 명령을 처리하기 위한 어드민 (Admin) 큐와 스토리지 I/O 명령을 처리하기 위한 일반 (Normal) 큐로 구분된다. 큐는 또다시 제출 큐 (submission queue)와 완료 큐 (completion queue)로 구분된다. 제출 큐는 호스트 시스템이 커맨드가 포함된 엔트리를 제출하고, 스토리지가 해당 엔트리를 가져와서 처리한다. 완료 큐는 스토리지가 완료 결과 엔트리를 제출하고, 스토리지가 해당 엔트리를 가져와서 처리한다. 큐의 프로듀서는 새 엔트리를 테일 (tail)에 추가하고, 컨슈머는 헤드 (head)부터 엔트리를 확인하기 시작해서 tail 까지 증가하면서 새 엔트리를 검사한다. 최대로 생성 가능한 큐의 개수는 64 K 이며, 큐의 최대 깊이 또한 64 K 이다. 또한

큐 간의 상대적인 중요도 (weight)를 설정함으로써 QoS 개념도 적용할 수 있다.

2.1.2 모바일 스토리지 인터페이스

모바일 시스템을 위한 스토리지는 모바일 시스템의 제약을 고려하여 저전력과 공간 효율성을 위주로 개발되었다. 모바일 시스템을 위한 다양한 스토리지 인터페이스가 존재하지만 여기서는 모바일 시스템에 내장되어 사용되는 eMMC와 UFS 인터페이스를 살펴본다.

2.1.2.1 eMMC

eMMC는 현재 가장 많이 사용되는 모바일 스토리지 규격이다. 최신 버전은 5.1이고 최대 400 MB/s의 전송속도를 제공한다 [4]. eMMC가 널리 사용되게 된 이유 중 한 가지는 간단한 아키텍쳐 때문이다. eMMC는 오직 12개의 핀을 이용해서 호스트 시스템과 통신한다; 8핀은 데이터, 1핀은 명령, 그리고 3개의 핀은 클록을 포함한 나머지 용도로 사용한다.

또한 eMMC는 *boot mode*라고 불리는 모바일 시스템에서 매우 유용한 기능을 제공한다. 호스트 시스템은 NOR 플래시 메모리나 EEPROM과 같은 별도의 장치 없이도 *boot mode*를 이용하면 eMMC만을 이용해서 부팅할 수 있다.

가장 최신 규격인 5.1에서는 NAND 플래시 메모리의 병렬성을 최대한 활용함으로써 스토리지의 성능을 극대화하기 위해서 *command queueing* 기능이 추가되었다. 이 기능을 이용하면 호스트 시스템이 동시에 여러 개의 명령을 스토리지에 이슈할 수 있다.

eMMC의 버전 별 최대 전송속도는 표 2에 정리되어 있다.

Version	Throughput (MB/s)
4.41	104
4.5	200
5.0	400

표 2: eMMC 버전별 최대 전송속도

2.1.2.2 UFS

UFS는 가장 최근에 제안된 모바일 스토리지 인터페이스이며, 비교적 최근에 상용화에 성공하였다. 최신 버전은 2.0 이며 현재 가장 빠른 모바일 스토리지 인터페이스로서 최대 전송속도는 1,200 MB/s (2개의 6 Gb/s 레인 사용 시) 이다 [5]. eMMC와 마찬가지로 *boot mode*와 *command queueing*을 지원한다.

은 그림 3은 UFS 의 전체 아키텍쳐를 보여준다. UFS가 eMMC와 구분되는 큰 특징 중 하나는 eMMC와 달리 시리얼 방식의 통신을 사용한다는 것이다. 또 다른 특징으로는 NVMe와 마찬가지로 스토리지가 호스트 시스템의 메모리에 접근할 수 있다는 것이다. 이 기능은 UME (unified memory extension) 인데 스토리지가 호스트 시스템의 메모리를 할당받아 스토리지가 필요한 용도로 사용할 수 있다 [6]. 일반적으로 모바일 스토리지는 비용, 공간, 전력소모 등의 원인으로 인해 리소스의 제약이 많기 때문에 동작에 필요한 최소한의 리소스만 갖추고 있는 경우가 많다. 따라서 메모리도 최소한의 용량만 갖추고 있는 경우가 많다. 만약 호스트 시스템의 메모리를 스토리지가 사용할 수 있다면 호스트 메모리를 펌웨어의 메타데이터를 올려두거나 미리 읽기 버퍼, 쓰기 버퍼 등으로 활용 가능하다. 한 연구에서는 메타데이터를 호스트 시스템 메모리에 로딩함으로써 성능을 대폭 향상시킬 수 있음을 증명하였다 [7].

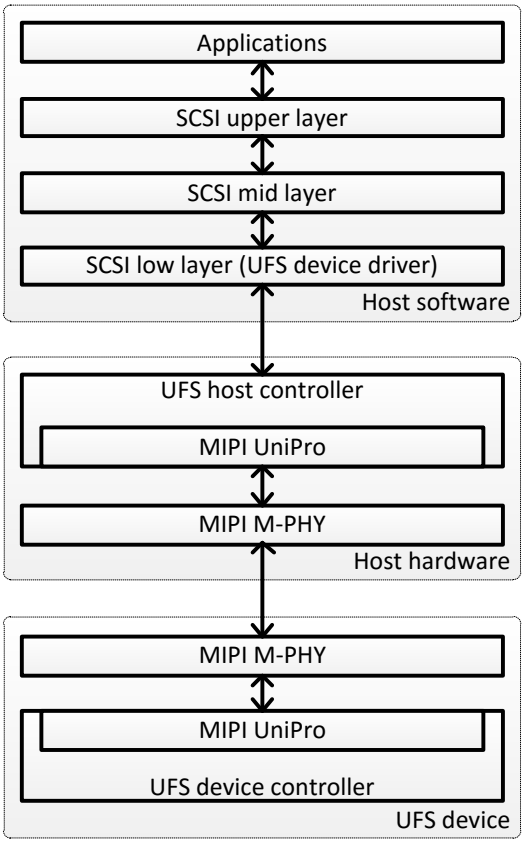
Applications
SCSI upper layer
SCSI mid layer
SCSI low layer (UFS device driver)
Host software
UFS host controller
MIPI UniPro
MIPI M-PHY
Host hardware
MIPI M-PHY
MIPI UniPro
UFS device controller
UFS device

그림 3: UFS 아키텍쳐

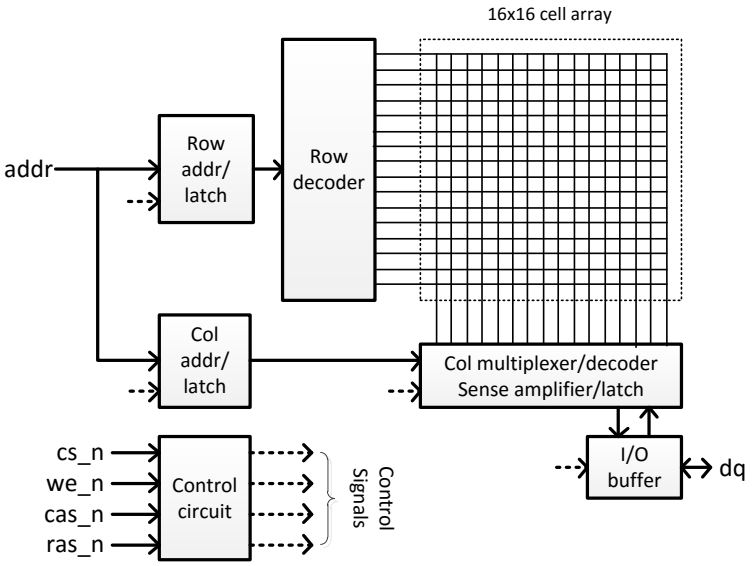

그림 4: DRAM 셀 구조

2.2 DRAM

DRAM은 바이트 단위 (또는 캐시 라인 단위)의 접근이 가능하고 속도가 매우 빠르며 SRAM 에 비해 상대적으로 가격이 저렴하여 주로 컴퓨터 시스템의 메인 메모리로 사용된다. 그러나 시간이 지나면 저장된 데이터가 소실되고, 이를 마기 위해 주기적으로 리프레시를 수행해야 하기 때문에 이로 인한 소비전력이 필요한 단점도 있다. 사용 환경에 따라 DDR [8], LPDDR [8], GDDR [8] 등으로 구분되나 기본 동작은 유사하다. 여기서는 DRAM 구조와 특성, 오퍼레이션, DRAM DIMM에 대해 살펴본다.

2.2.1 DRAM 구조와 특성

DRAM은 그림 4과 같은 구조로 구성된다 [9]. 하나의 DRAM 셀은 DRAM 스위칭 트랜지스터 한개와 캐패시터 한 개로 이루어진다. 셀의 상태 (0 또는 1)는 캐피시터의 상태에 의해 결정되며, 여러 개의 셀이 모여서 하나의 셀 어레이를 구성한다.

셀의 데이터 입출력은 다음과 같이 진행된다. 셀의 데이터를 읽기 위해서는 row 디코더와 column 디코더, 센스 앰프를 사용한다. 먼저 row 디코더를 통해 대상 row 의 셀을 활성화 (activate) 한다. 그 후 column 디코더를 이용해 대상 row 셀 중 타겟 clumn의 셀을 활성화 (activate) 한다. 이제는 센스 앰프를 이용해서 최종 선택된 셀의 데이터를 읽어낸다. 셀에 데이터를 쓰는 방법도 유사한 순서를 통해 진행한다.

셀 어레이가 모여서 뱅크 (bank), 랭크 (rank)를 구성하게 되며, 하나의 DIMM 은 다수의 랭크로 구성될 수 있다. 하나의 채널에는 다수의 DIMM 을 연결할 수 있으나, 채널은 다수의 DIMM에 의해 공유되므로 호스트의 메모리 컨트롤러는 특정 시점에서 하나의 DIMM에 대해서만 I/O를 진행할 수 있다.

일반적으로 DRAM 서브시스템은 다수의 뱅크, 랭크, 채널로 구성되기 때문에 병렬성을 제공한다. DRAM 컨트롤러는 이러한 병렬성을 활용하여 인터리빙 기법을 적용함으로써 DRAM 서브시스템의 성능을 개선할 수 있다 [10]. 그러나 본 연구에서 제안한 스토리지 인터페이스는 DRAM으로 위장한 메모리 장치이므로, 기존의 시스템에서 사용하던 인터리빙 기법을 사용하지 않는다. 오히려 기존의 인터리빙 기법을 필요에 의해 설정할 수 없는 점이 문제점으로 부각되는데, 이에 대해서는 6.3.2 유연한 DRAM 주소 매핑에서 자세히 살펴본다.

2.2.2 DRAM 오퍼레이션

- Read

  Read 오퍼레이션은 ACT 명령과 함께 row 주소를 전송한 후, t_{RCD} 이후에 READ 명령과 함께 col 주소를 전송한다. CL 시간 이후 DATA 시그널을 통해 읽기 데이터가 전송된다. 읽기 데이터 전송이 완료되면 t_{RP} 시간 동안 precharge를 수행한다.

- Write

  Write 오퍼레이션은 ACT 명령과 함께 row 주소를 전송한 후, t_{RCD} 이후에 WRITE 명령과 함께 col 주소 및 쓰기 데이터를 전송한다. 쓰기 데이터 전송 완료 후 t_{WR}이후 t_{RP} 동안 row precharge를 수행한다.

2.2.3 DRAM DIMM

일반적으로 데스크탑/서버 시스템에서는 DRAM을 다수의 DRAM 칩이 PCB에 장착된 DIMM (Dual Inline Memory Module) 형태로 사용한다. 데스크탑 시스템에서는 같은 비용으로 큰 용량의 DRAM을 사용하는 것이 가장 중요하므로, 가장 단순한 형태의 UDIMM을 주로 사용한다. 그러나 서버 시스템에서는 안정성과 용량 확장을 위해 ECC가 포함된 RDIMM 또는 LRDIMM을 사용한다.

- UDIMM

  UDIMM (unbuffered DIMM)은 그림 5의 (a)와 같은 간단한 구조로 이루어져있다. ECC를 지원하는 경우 ECC 패리티를 저장하기 위한 DRAM 칩이 추가로 장착된다. UDIMM의 DRAM 칩은 호스트 시스템의 DRAM 컨트롤러와 명령, 주소, 데이터 시그널이 직접 연

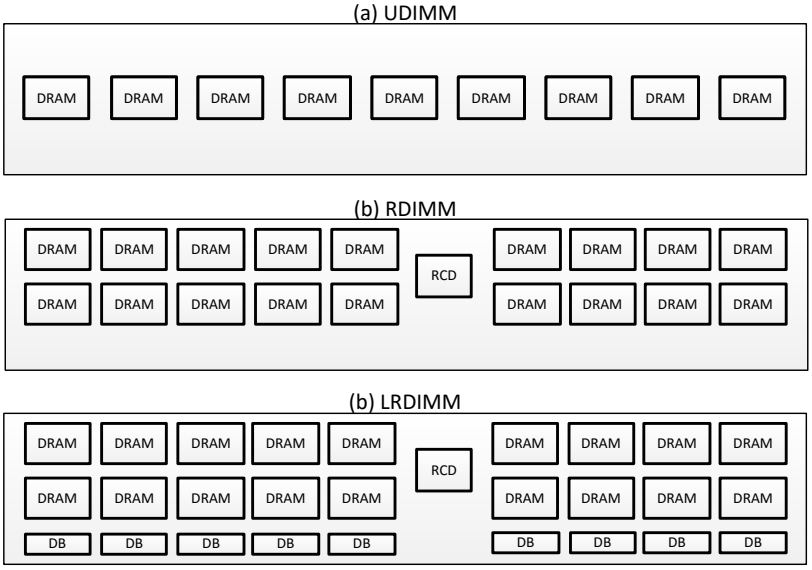

그림 5: DRAMM DIMM 분류

결된다. 따라서 DRAM 칩 이외의 부품 사용을 최소화하여 가격이 저렴한 장점이 있다.

그러나 DRAM 칩이 호스트 시스템의 DRAM 컨트롤러와 직접 연결되므로 시그널 품질 문제 때문에 다수의 칩을 장착하기가 어려운 문제가 있다. 이 문제를 해결하기 위해 다음에서 기술하는 RDIMM, LRDIMM 을 사용한다.

- RDIMM

RDIMM (registered DIMM)은 그림 5의 (b)와 같은 구조를 사용하며 명령과 주소 시그널의 품질을 개선하기 위해 RCD (Registering Clock Driver)를 사용한다. RCD는 DIMM의 중앙에 위치하며 호스트 시스템의 DRAM 컨트롤러에서 DRAM 명령과 타겟 주소를 전달

받아서 RDIMM에 있는 각각의 DRAM 칩에 분배한다. 이 때 RCD와 각 DRAM 칩 까지의 거리가 다르기 때문에 이 거리차를 고려하여 DRAM 칩에 명령과 주소를 전달함으로서 결과적으로 RDIMM의 모든 DRAM 칩이 같은 시점에 명령과 주소를 전달받을 수 있도록 한다. 따라서 호스트 시스템의 DRAM 컨트롤러와 DRAM 칩이 직접 연결되는 UDIMM에 비해 명령과 주소 시그널의 품질이 개선되므로 DIMM에 보다 많은 DRAM을 장착할 수 있다. 그러나 RCD 칩을 추가하는데 비용이 필요하며, 여전히 데이터 시그널은 호스트 시스템의 메모리 컨트롤러와 직접 연결되므로 DRAM 칩의 개수를 늘리는데 한계가 있다.

- LRDIMM

LRDIMM (load reduced DIMM)은 RDIMM보다 더 많은 용량의 DRAM 칩을 장착하기 위해 사용한다. RDIMM에서는 RCD가 각 DRAM 칩까지 명령과 주소 시그널을 동시에 전달하는 역할을 담당하지만, 데이터 시그널은 여전히 호스트 시스템의 메모리 컨트롤러와 DRAM 칩이 직접 연결되므로 시그널의 길이 차이로 인한 문제가 발생한다. 이 문제를 개선하기 위해 LRDIMM은 그림 5의 (c)와 같이 DIMM 내부에 해당 데이터를 버퍼링하는 버퍼 칩을 추가하고, DRAM에서 읽거나 저장할 데이터를 해당 버퍼에 임시로 저장해두었다가 호스트 시스템으로 전송하거나 DRAM으로 저장한다. 따라서 호스트 시스템 관점에서는 LRDIMM 내부에 다수의 DRAM 칩이 존재하더라도 데이터 시그널에 대해서도 동일한 타이밍을 얻게 되는 장점이 있다. 그러나 버퍼 칩을 추가하는데 비용이 필요하며, DRAM에 입출력할 데이터가 무조건 버퍼 칩을 통과해야 하므로 레이턴시가 증가하는

단점이 있다.

이와 같이 현재 다양한 유형의 DIMM이 존재한다. 호스트 시스템은 이와 같은 다양한 유형, 용량, 속도를 갖는 DRAM DIMM 지원해야 한다. 이를 위해서 바이오스와 같은 시스템 초기화 소프트웨어는 시스템에 장착된 DRAM DIMM의 정보를 확인하고 적절한 설정을 진행하는데, 이를 위해 DIMM의 SPD (Serial Presence Detector)를 사용한다. SPD 정보는 DRAM DIMM의 별도의 EEPROM에 저장되어 있으며 DRAM의 용량, 속도, 제조사 등에 대한 정보를 포함하고 있다. SPD 정보는 DRAM 인터페이스가 활성화되기 전에 필요하기 때문에 호스트 시스템은 매우 간단한 프로토콜인 I2C를 이용해서 DRAM DIMM으로부터 SPD 정보를 읽은 후 이 정보를 이용해서 DRAM DIMM 관련 설정을 진행한다.

2.3 NAND 플래시 메모리

NAND 플래시 메모리는 탐색 시간이 없고, 병렬성을 활용한 성능 향상에 유리하며, 충격에 강하기 때문에 매우 광범위한 분야에서 저장 매체로 사용되고 있다. 여기서는 NAND 플래시 메모리의 구조와 특성, 오퍼레이션, 그리고 NAND 플래시 메모리를 위한 소프트웨어에 대해 살펴본다.

2.3.1 NAND 플래시 메모리의 구조

NAND 플래시 메모리는 크게 NAND 플래시 메모리 셀 어레이와 외부와의 데이터 교환을 담당하는 레지스터 (또는 페이지 버퍼)로 나눌 수 있다. 셀 어레이의 구조는 그림 6과 같으며, 페이지가 모여 블록이 된다. 하나의 NAND 플래시 메모리 칩에 포함되어 있는 모든 블록은 하나의 레지스터 (또는 페이지 버퍼)를 공유하며, 반드시 이 레지스터를 통해서만

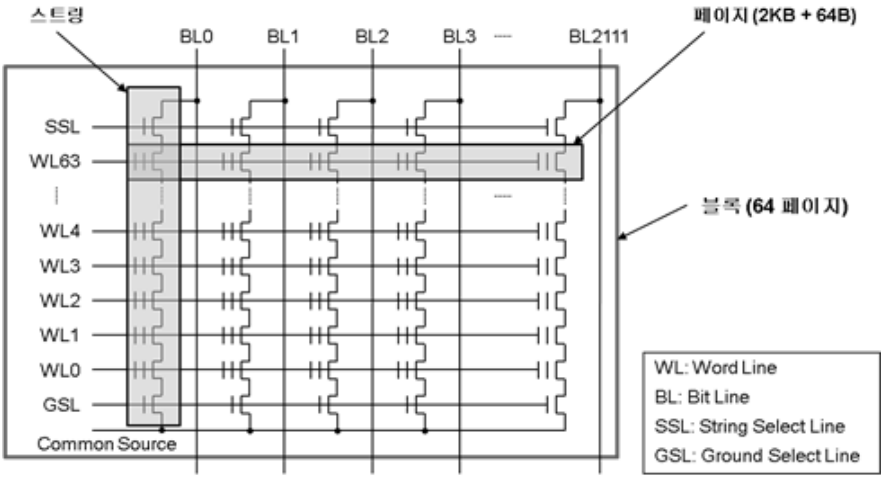

그림 6: NAND 플래시 메모리 셀 어레이 구조

어레이 외부와 데이터를 교환할 수 있다.

- 셀

  셀 (cell)은 데이터를 저장하는 기본 단위이다. 집적도에 따라 SLC (single level cell), MLC (multi level cell), TLC (triple level cell)로 구분되는데 각각 셀 당 1, 2, 3 비트를 저장한다.

  셀의 상태는 SLC 기준으로 1 (erased) 또는 0 (programmed)로 구분되며, MLC는 4가지 상태, TLC는 8가지 상태를 갖는다. 셀의 상태는 erase 오퍼레이션을 통해 0에서 1로 천이할 수 있고, program 오퍼레이션을 통해 1에서 0으로 천이할 수 있다.

- 페이지

  페이지 (page)는 셀로 구성되며 Read와 Program 오퍼레이션의 기본 단위다. SLC는 한 셀이 하나의 페이지만 속하지만, MLC, TLC의 경우는 한 셀이 여러 페이지에 속할 수 있다. 하나의 페이지는 데이터

영역과 스페어 여역으로 나뉜다. 그림 6의 NAND 플래시 메모리는 페이지의 크기를 2112 바이트로 가정하고 있다. 여기서 실제 데이터를 저장하는 데이터 영역은 2048 바이트이고, 나머지 64 바이트 영역은 스페어 영역으로 불리는데 여기는 데이터에 대한 ECC나 기타 필요한 정보를 저장하는 용도로 사용한다.

- 블록

  블록 (block)은 Erase 오퍼레이션의 기반 단위이며, 여러 개의 페이지가 모여 하나의 블록을 구성한다. 그림 6의 NAND 플래시 메모리는 하나의 블록이 64개의 페이지로 구성되는 것을 가정한다. 이 경우 하나의 블록의 크기는 데이터 영역만 계산하면 128 KB이고, NAND 플래시 메모리 칩 하나가 2048 개의 블록으로 구성된다고 가정하면 칩의 용량은 256 MB, 즉 2 Gb 이다.

  NAND 플래시 메모리의 주요한 특성 중 한 가지는 배드 블록이 발생할 수 있다는 것이다. 배드 블록은 양산 과정에서 발생한 초기 배드 블록과 동작중에 발생하는 런타임 배드 블록으로 구분할 수 있는데, NAND 플래시 메모리 소프트웨어는 이러한 배드 블록을 사용하지 않고 정상 블록만 사용해야 한다.

2.3.2 NAND 플래시 메모리 오퍼레이션과 인터페이스

앞에서 NAND 플래시 메모리의 구조를 살펴봤으므로, 이제는 NAND 플래시 메모리의 주요 오퍼레이션인 Read, Program, Erase 에 대해 살펴보자.

- Read

Read 오퍼레이션은 페이지 단위로 동작하며, 셀 어레이에 저장된 데이터를 페이지 버퍼로 읽어온다. 한 셀에 한 비트의 데이터만 저장되는 SLC에 비해 2, 3 비트의 데이터가 저장되는 MLC와 TLC는 SLC에 비해 데이터를 읽는 시간이 더 오래 걸린다. 페이지 버퍼에 읽어온 데이터는 스페어 영역에 저장된 ECC 정보를 활용해서 에러 탐지 및 정정 과정을 거쳐서 원본 데이터로 복원해야 한다.

그림 6에서 볼 수 있듯이 NAND 플래시 메모리 셀은 서로 인접하고 있기 때문에 한 페이지를 읽더라도 주변 페이지에 영향을 주 수 있다. 이 문제를 읽기 간섭 (read disturb)이라고 하며 특정 페이지쳐를 반복해서 읽는 경우 이 문제가 발생하지 않도록 소프트웨어적인 조치가 필요하다.

- Program

Program 오퍼레이션도 페이지 단위로 동작하며, 페이지 버퍼의 데이터를 셀 어레이에 저장한다. 이 오퍼레이션은 NAND 플래시 메모리의 셀 특성으로 인해 반드시 erase 가 완료된 페이지에만 수행해야 한다. 프로그램 오퍼레이션 수행 중에 에러가 발생할 수 있으므로 이에 대한 적절한 처리가 필요하다.

하나의 셀에 2, 3 비트가 저장되는 MLC, TLC의 경우는 한 셀이 LSB (least significant bit), MSB (most significant bit), CSB (TLC의 경우) 페이지에 의해 공유된다. 따라서 특정 셀이 포함되는 LSB 페이지에 프로그램이 완료되더라도, MSB 페이지에 프로그램 수행중에 전원 중단이나 프로그램 에러로 인해 프로그램이 중단되는 경우 이미 프로그램이 완료된 LSB 페이지의 데이터도 손상될 수 있다. 이 문제를 페어드 페이지 문제 (paired page problem)이라고 부르며, 이에 대한

대비가 필요하다.

- Erase

  Erase 오퍼레이션은 블록 단위로 동작하며, 블록의 상태를 erase로 초기화한다. Program 오퍼레이션의 설명에서 기술한 것과 같이 erase 된 페이지에 대해서만 프로그램을 수행할 수 있다. 일반적으로 erase 시에 에러가 불생하는 블록은 불량 블록으로 간주해 더이상 사용하지 않도록 처리한다.

NAND 플래시 메모리는 그림 7와 같은 인터페이스를 사용한다. 각 인터페이스 핀의 용도는 다음과 같이 크게 세 가지로 구분된다: ① 컨트롤러가 NAND 플래시 메모리에 명령과 주소를 전달 ② NAND 플래시 메모리가 자신의 상태를 컨트롤러에 전달 ③ NAND 플래시 메모리에 쓸 데이터, 또는 NAND 플래시 메모리에서 읽은 데이터 전송.

과도기적인 NAND 플래시 메모리 제품 중에는 NAND 플래시 메모리 뿐 아니라 SRAM을 내장하고 있어 해당 SRAM을 통해 NAND 플래시 메모리를 제어할 수 있는 제품도 있었다. 해당 제품은 OneNAND 라는 제품명으로 상용화되었고, 그림 8처럼 임베디드 시스템에서 부팅에 필요한 데이터가 저장된 NOR 플래시 메모리를 제거할 수 있는 효과가 있어 한동안 널리 사용되었다. 또한 NAND 플래시 메모리의 인터페이스가 변경되더라도 SRAM 영역에 정의된 NAND 플래시 메모리를 제어하기 위한 인터페이스는 변경되지 않으므로 신규 NAND 플래시 메모리 제품도 호스트 시스템의 변경 없이 사용할 수 있는 장점이 있다. 그러나 SRAM 및 ECC를 포함한 관련 로직의 추가로 인해 NAND 플래시 메모리 단품에 비해 가격 경쟁력이 떨어져서 최근에는 사용되지 않고 있다.

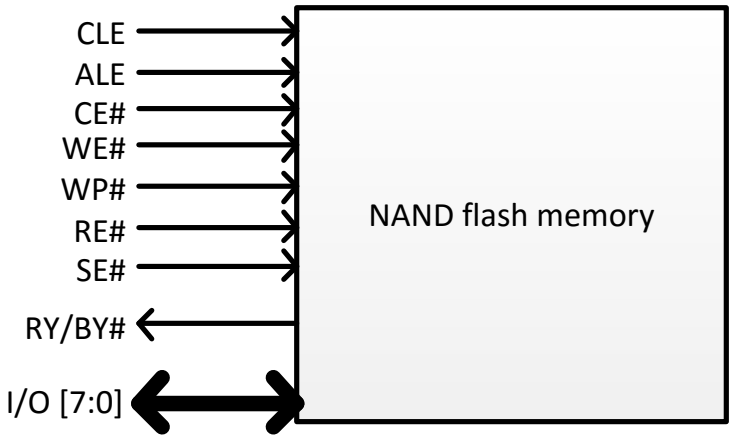

그림 7: NAND 플래시 메모리 인터페이스

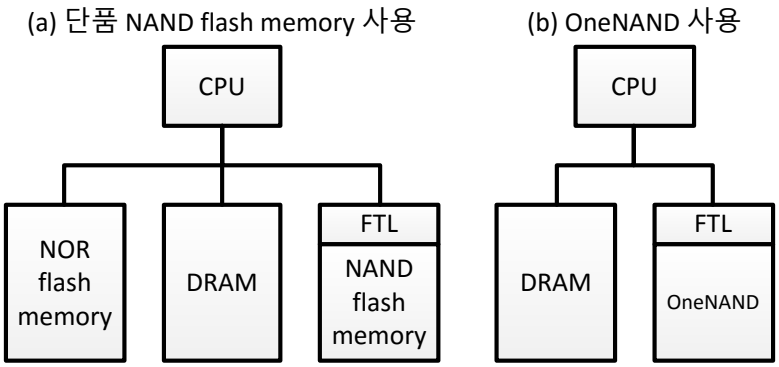

그림 8: OneNAND 아키텍쳐

2.3.3 플래시 변환 계층

앞에서 살펴본 것과 같이 NAND 플래시 메모리의 read, write 오퍼레이션 단위는 일반적인 스토리지 인터페이스의 섹터 (512 바이트)가 아닌 페이지이다. 따라서 NAND 플래시 메모리를 스토리지로 사용하려면 섹터 주소를 NAND 플래시 메모리 주소로 변환해주는 소프트웨어 계층이 필요하며, 이것을 플래시 변환 계층이라고 한다. 또한 플래시 변환 계층은 NAND 플래시 메모리의 덮어쓰기가 불가능한 특성 때문에 발생하는 erase 오퍼레이션 및 가비지 컬렉션, NAND 플래시 메모리의 배드 블록을 사용하지 않도록 하는 역할을 담당한다.

- 주소 변환

  플래시 변환 계층의 가장 기본적인 역할 중 하나로, 스토리지 인터페이스의 오퍼레이션 단위인 섹터 (논리 주소)와 NAND 플래시 메모리의 오퍼레이션 단위인 페이지 또는 블록 (물리 주소)간의 매핑을 관리한다. 호스트가 read를 요청하면, ① 대상 섹터 주소에 대한 NAND 플래시 메모리의 물리 주소를 확인하고 ② 해당 물리 주소에서 데이터를 읽어서 ③ 호스트에 반환한다. 호스트가 write를 요청하면, ① erase된 페이지 (또는 블록)에 호스트의 write 데이터를 프로그램하고 ② 논리-물리 주소의 매핑 정보를 새 물리 주소로 업데이트하고 ③ 호스트에 write 결과를 반환한다.

  논리주소를 물리주소로 매핑하는 단위에 따라 페이지 매핑, 블록 매핑, 하이브리드 매핑 등이 있다. 페이지 매핑은 랜덤 쓰기 성능이 좋지만 매핑 정보가 큰 단점이 있다. 블록 매핑은 매핑 정보의 크기가 작지만 랜덤 쓰기 성능이 매우 낮은 단점이 있다. 하이브리드 매핑은 페이지 매핑과 블록 매핑의 장점을 합한 것으로 매핑 정보

가 작으면서도 성능이 좋지만, 매핑 정보의 관리가 복잡한 문제가 있다. 본 논문의 이후 내용에서는 페이지 매핑을 가정하고 논의를 진행한다.

- 가비지 컬렉션

  덮어쓰기가 불가능한 NAND 플래시 메모리의 특성 때문에 호스트의 write 데이터는 erase 된 페이지에 쓰여지고, 해당 섹터에 매핑되어 있던 NAND 플래시 메모리 페이지의 내용은 무효화된다. 하나의 스토리지 내부에 있는 NAND 플래시 메모리 블록의 개수는 한정되어 있으므로, 계속해서 호스트의 write를 처리하면 자유 블록이 점차 소진되고, 더이상 자유 블록이 없으면 호스트의 write 처리가 불가능하다. 따라서 프로그램이 완료된 블록 중 대상 블록을 선정해서 유효한 페이지는 다른 블록으로 복사한 후 대상 블록을 erase 함으로써 자유 블록을 지속적으로 확보해야 한다. 이렇게 빈 블록을 확보하는 과정을 가비지 컬렉션이라고 한다.

  가비지 컬렉션은 NAND 플래시 메모리 기반 스토리지에서는 반드시 필요하지만 스토리지의 성능과 수명에 영향을 주기 때문에 이를 최소화할 수 있어야 한다. 호스트의 요청을 처리하면서 가비지 컬렉션을 수행하면 호스트에게 제공할 수 있는 스토리지의 성능이 떨어지므로 가비지 컬렉션은 가능한 유휴 상태에서 진행하는 것이 좋다. 또한 호스트가 한 번 write 한 데이터를 다른 위치에 write 하는 작업이기 때문에 필연적으로 erase와 program을 발생시켜 NAND 플래시 메모리의 수명을 감소시킨다. 이렇게 가비지 컬렉션으로 인해 더 쓰여지는 양을 수치화 하기 위해 WAF (write amplification factor), WAI (write amplification index) 등이 정의되어 가비지 컬렉션 정책

의 효율을 평가하는데 사용된다.

가비지 컬렉션 정책의 효율을 결정하는 가장 중요한 한 가지는 타겟 블록을 선정하는 방법이며, 이에 대해 많은 연구가 진행되었다. 일반적으로는 해당 블록의 유효 페이지가 가장 적은 블록을 선정하는 방법이 사용되는 것으로 알려져 있다.

- 배드 블록 관리

  배드 블록을 사용가능한 블록에서 제외시킴으로써 배드 블록의 사용을 막는 것도 플래시 변환 계층의 역할이다. 이것은 NAND 플래시 메모리가 NOR 플래시 메모리와 달리 배드 블록의 발생을 허용하기 때문이다. 배드 블록은 제조 공정에서 이미 불량인 초기 배드 블록과 사용중에 발생하는 런타임 배드 블록의 두 종류가 있다. 플래시 변환 계층이 처음 실행되어 초기화될때 초기 배드 블록을 검색해서 사용하지 않도록 한다. 초기 배드 블록은 이를 식별하기 위한 정보가 해당 블록에 마킹되어 있어 해당 정보를 읽어봄으로써 확인이 가능하다. 런타임 배드 블록은 다시 두 가지로 나눌 수 있는데 erase 오퍼레이션이 실패하는 배드 블록과 program 오퍼레이션이 실패하는 배드 블록으로 분류할 수 있다. 따라서 플래시 변환 계층은 erase와 program 오퍼레이션이 실패하는 경우 해당 블록을 배드 블록으로 등록함으로써 더이상 사용되지 않도록 해야 한다. 특히 program 오퍼레이션이 실패하는 경우 해당 블록에 이미 program 된 기존의 데이터는 가비지 컬렉션을 이용해서 다른 정상 블록으로 옮겨야 한다.

- 웨어 레벨링

  NAND 플래시 메모리는 정상 동작을 보장하는 P/E (program/erase)

횟수가 이 제조사에 의해 정해져 있으며, 이 횟수를 넘어가면 정상 동작을 보장하지 않는다. 스토리지는 사양에서 지정된 용량을 제공해야 하는데, 만약 스토리지를 구성하는 NAND 플래시 메모리의 일부 블록의 P/E 횟수가 급격히 증가하여 지정된 용량을 제공할 수 없으면 이 스토리지는 더이상 사용이 불가능해진다. 따라서 일부 NAND 플래시 메모리 블록만 다른 블록에 비해 P/E 횟수가 빠르게 증가하여 웨어 아웃 (wear out)됨으로써 스토리지를 더이상 사용할 수 없는 상황을 막아야 한다. 이를 막기 위해서는 전체 플래시 메모리 블록의 P/E 횟수를 균등하게 유지할 필요가 있으며, 이를 위한 기법을 웨어 레벨링이라고 한다.

웨어 레벨링은 크게 소극적인 (reactive) 웨어 레벨링과 적극적인 (proactive) 웨어 레벨링으로 나눌 수 있다. 소극적인 웨어 레벨링은 웨어 레벨링 자체를 위해 데이터를 이동시키지 않는다. 대신 호스트의 write 데이터나 가비지 컬렉션의 write 데이터를 저장하는 대상 블록을 선정할 때 개입하여 자유 블록 중 P/E 횟수가 낮은 블록을 선택하게 함으로써 해당 블록의 P/E 횟수가 자연스럽게 증가하도록 한다.

반면 적극적인 웨어 레벨링은 웨어 레벨링을 자체를 위해 데이터를 이동시키기도 한다. 예를 들면 특정 블록에 한번 호스트가 write 한 후 오랫동안 업데이트가 이루어지지 않는 데이터가 있다면, 이 블록은 유효 페이지 개수가 많아서 가비지 컬렉션 대상 블록에서도 제외될 가능성이 높다. 따라서 이 블록은 오랫동안 erase 되지 않고 유지되어 다른 블록에 비해 상대적으로 P/E 회수가 낮게 유지될 가능성이 높다. 이 상황에서는 웨어 레벨링을 통해 적극적으로 개입

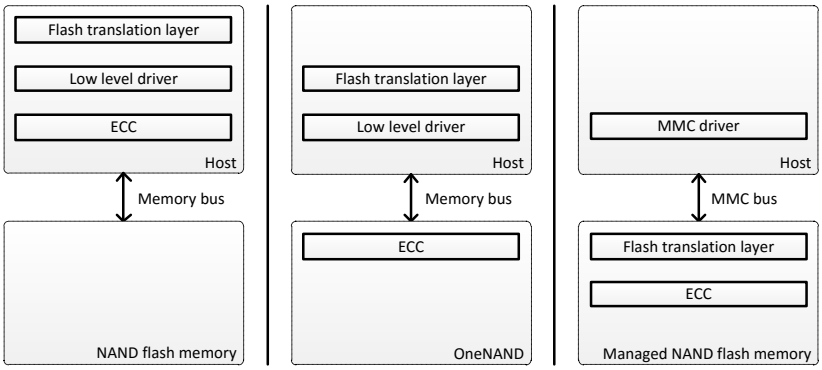

그림 9: 플래시 변환 계층의 위치

하지 않으면 계속 유지될 가능성이 높기 때문에, 적극적인 웨어 레벨링 기법은 해당 블록의 데이터를 강제로 다른 블록으로 (주로 P/E 횟수가 높은) 이동시킨다. 특히 해당 데이터가 이동된 블록이 P/E 횟수가 높은 경우에는 해당 데이터에 대한 업데이트가 발생하지 않아서 해당 블록의 P/E 사이클이 그대로 유지되는 효과도 기대할 수 있다.

이와 같은 플래시 변환 계층은 그림 9에서와 같이 호스트 시스템 또는 스토리지 내부에서 실행될 수 있다. NAND 플래시 메모리의 도입 초기에는 호스트 시스템에서 플래시 변환 계층을 실행하면서 호스트 시스템이 NAND 플래시 메모리를 직접 관리하는 형태로 사용되었다. 그러나 NAND 플래시 메모리가 급격히 발전하면서 NAND 플래시 메모리를 관리하는 오버헤드가 늘어남에 따라 최근에는 주로 스토리지 장치에서 플래시 변환 계층을 실행하여 블록 인터페이스를 제공하는 형태로 사용되고 있다.

2.3.4 플래시 파일 시스템

NAND 플래시 메모리 도입 초기에는 별도의 플래시 변환 계층을 사용하지 않고, 호스트 시스템에서 플래시 메모리 전용 파일 시스템을 통해 직접 NAND 플래시 메모리를 관리하는 형태로 사용했다. 이러한 파일 시스템을 일반적으로 플래시 파일 시스템 (flash file system) 이라고 부른다. 플래시 메모리 파일 시스템은 일반적으로 플래시 변환 계층이 수행하는 작업을 수행하면서 상위 계층에는 일반적인 파일 시스템 인터페이스를 제공한다. 플래시 파일 시스템에는 다음과 같은 것들이 있다.

- JFFS2

  JFFS2 (journaling flash file system version 2)는 로그 스트럭쳐 기반의 플래시 파일 시스템이다 [11]. 초기 버전인 JFFS는 NOR 플래시 메모리를 위해 개발되었으나, 다음 버전인 JFFS2는 NAND 플래시 메모리를 지원한다. 파일과 디렉토리의 변경사항은 *inode*와 *dirent node*에 로깅되고, 파일과 디렉토리에 업데이트가 발생하면 새 노드가 생성되고 기존의 노드는 무효화된다. JFFS2의 가장 큰 단점은 로그 스트럭쳐 기반 파일 시스템에서 일반적으로 발생하는 긴 마운트 시간이다. 로그 스트럭쳐 기반 파일 시스템은 모든 변경사항을 로그 형태로 기록한다. 따라서 파일 시스템이 마운트 될 때 최신 상태를 확보하기 위해서는 로그 전체를 읽어야 하는 문제가 있다. 이를 해결하기 위해서 JFFS2는 EBS (erase block summary)를 각 NAND 플래시 메모리 블록의 끝 부분에 기록하고, 마운트시에는 해당 정보만을 읽음으로써 마운트 시간을 단축한다.

- YAFFS

YAFFS (yet another flash file system)는 JFFS와 다르게 처음부터 NAND 플래시 메모리를 위해 작성된 파일 시스템이며 현재 버전 2까지 개발되어 있다 [12] [13]. YAFFS1은 페이지 크기가 512 바이트인 NAND 플래시 메모리를 지원하며, YAFFS2는 페이지 크기가 2048 바이트인 NAND 플래시 메모리도 지원한다. YAFFS는 파일의 데이터를 기록할 때, 가장 처음에는 파일의 메타데이터를 기술하는 페이지를 기록한다. 새 파일이 생성될 때 마다 고유한 *object ID*를 할당받는데, 한 파일에 포함된 모든 데이터는 이 *object ID*를 이용해서 관리된다. 또한 새로 쓰여지는 모든 블록에 대해 일련 번호를 표시하는데, 이것은 단조 증가한다. 따라서 파일 내의 같은 오프셋을 가리키는 데이터가 여러 개 존재하더라도 일련 번호를 활용해 최신 데이터를 식별할 수 있다. 또한 체크포인팅 기법을 활용함으로써 마운트 과정에서 불필요한 스캔 작업을 제거하고 스캔 시간을 대폭 단축했다.

- UBIFS

  UBIFS (unsorted block image file system)는 Linux의 UBI (unsorted block image) 장치 위에서 동작하는 플래시 파일 시스템이다 [14]. UBI는 NAND 플래시 메모리의 배드 블록에 대한 리매핑과 웨어 레벨링을 수행하므로 UBIFS는 NAND 플래시 메모리가 정상 블록인 것을 가정하고 동작할 수 있다. UBIFS는 쓰기 캐싱을 지원하기 때문에 JFFS2와 비교하여 쓰기 속도가 빠르고, 파일 시스템 인덱스를 NAND 플래시 메모리에 저장하기 때문에 마운트 속도가 빠르다.

- F2FS

  F2FS (flash friendly file system)은 NAND 플래시 메모리를 위한 파

일 시스템이지만, 앞에서 설명한 플래시 파일 시스템과는 달리 일반적인 블록 인터페이스 기반으로 동작한다 [15] [16]. 일반적으로 NAND 플래시 메모리 스토리지에서의 임의 쓰기는 스토리지의 성능과 수명을 저하시키는 것으로 알려져 있다. 그러나 기존의 파일 시스템은 이러한 NAND 플래시 메모리의 특성을 고려하지 않고 임의 쓰기를 수행하므로 NAND 플래시 메모리에 최적화되어있지 않다.

F2FS는 기존 파일 시스템과 다르게 전체 스토리지 영역을 6개의 구역으로 나누고 해당 구역 안에서는 순차 쓰기만을 수행한다. 그리고 주기적으로 플래시 변환 계층에서 수행하는 것과 유사한 가비지 컬렉션 작업을 수행한다. 가비지 컬렉션이 수행되어 파일 시스템 수준에서 더이상 유효한 데이터가 존재하지 않는 영역에 대해서는 스토리지에 힌트를 전달하여 해당 영역에 대해 스토리지 내부에서 불필요한 가비지 컬렉션이 발생하지 않도록 방지한다. 로그 스트럭쳐 기반의 파일 시스템에서는 하나의 업데이트가 상위 계층의 로그에 대한 계속적인 업데이트를 발생시키는 *wandering tree* 문제가 존재한다. F2FS는 해당 문제를 *node address translation* 기법을 활용하여 해결했다.

2.4 NVDIMM 장치

NAND 플래시 메모리와 같은 비휘발성 메모리를 활용하여 데이터를 비휘발성으로 보존하는 DIMM을 일반적으로 NVDIMM (Non-Volatile DIMM)이라고 부르며, NVDIMM-N, F, P의 세 가지 유형으로 분류된다 [17] [18]. 또한 최근에 Intel에서 발표한 3D Xpoint 메모리로 구성된 Intel DIMM 또한 NVDIMM 으로 분류할 수 있으나, 상세한 정보가 공개되지

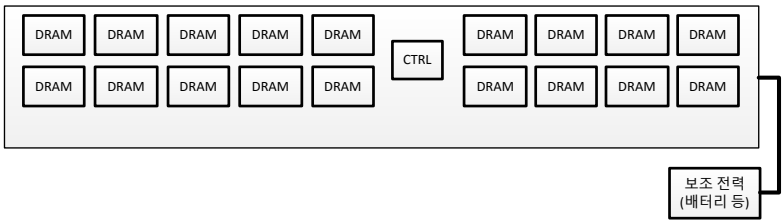

그림 10: NVDIMM-N 아키텍쳐

않아 어떤 NVDIMM 유형에 해당되는지는 정확한 판단이 어렵다.

2.4.1 NVDIMM-N

NVDIMM-N은 그림 10에서 보는 것과 같이 DRAM, NAND 플래시 메모리, 컨트롤러, 보조 전원 (배터리 등)으로 구성된다. 일반적인 상황에서는 DRAM만을 사용해서 DRAM DIMM과 동일하게 동작한다. 그러나 전원이 차단되면 NVDIMM-N 컨트롤러가 DRAM의 데이터를 같은 DIMM 내부에 있는 NAND 플래시 메모리로 백업한다. 데이터가 백업되는 동안은 호스트 시스템에서 전원을 공급받을 수 없기 때문에 배터리와 같은 보조 전원을 사용한다. 호스트 시스템에 다시 전원이 공급되면 NVDIMM-N 컨트롤러를 이용해서 NAND 플래시 메모리에 백업된 데이터를 DRAM에 리스토어함으로써 전원이 차단되기 전과 동일한 상태로 복구할 수 있다.

NVDIMM-N은 다음과 같은 장점을 갖기 때문에 점차 널리 사용되고 있다.

1. DRAM과 동일한 성능을 갖는 비휘발성 메모리

   NVDIMM-N의 가장 큰 장점이다. 일반적인 상황에서는 DRAM을

대상으로 입출력이 발생하므로 DRAM의 고성능을 그대로 활용할 수 있다. 상용화된 메모리 매체 중에서는 SRAM을 제외하면 DRAM이 가장 빠르기 때문에 최고의 성능과 비휘발성을 동시에 활용할 수 있다.

2. 반영구적인 수명

위와 마찬가지로 일반적인 상황에서는 DRAM을 대상으로만 입출력이 발생하므로 DRAM과 마찬가지로 반영구적인 수명을 갖는다. 단, DRAM의 데이터가 백업되는 NAND 플래시 메모리의 경우 수명이 있으나 NVDIMM-N 이 주로 사용되는 서버 환경에서는 전원 차단이 거의 발생하지 않기 때문에 문제가 없는 수준이다.

그 이유는 다음과 같다. NVDIMM-N 장치에 DRAM과 동일한 용량의 NAND 플래시 메모리가 장착되어 있고 (보통 DRAM 용량의 2-4배), 하루에 한번씩 전원 공급이 중단되어 NAND 플래시 메모리에 백업을 해야 한다고 가정하자. 1년을 사용하면 NAND 플래시 메모리의 Erase 회수는 365이고, 5년을 사용하더라도 1825회가 된다. NAND 플래시 메모리의 보장 Erase 회수는 제조사와 제품마다 다르지만, 일반적으로 수천 회를 보장하는 것으로 알려져 있다. 따라서 DRAM 용량과 NAND 플래시 메모리 용량이 1:1인 경우에도 5-10년 정도는 동작에 문제가 없을 것으로 예상된다. 게다가 NVDIMM-N의 NAND 플래시 메모리 용량은 일반적으로 DRAM 용량의 2-4배를 사용하고, 서버 환경에서는 전원 차단이 거의 발생하지 않는다는 점을 고려하면 NVDIMM-N의 NAND 플래시 메모리 수명은 NVDIMM-N을 본격적으로 활용하는데 문제가 없는 수준으로 판단된다.

3. 호스트 시스템 하드웨어 변경 불필요

NVDIMM-N은 기존의 DRAM 인터페이스와 100% 호환되므로 호스트 시스템의 하드웨어를 변경할 필요가 없다. 그것은 DRAM DIMM에는 없는 NVDIMM-N의 고유한 작업인 NAND 플래시 메모리를 이용한 백업과 리스토어가 NVDIMM-N 컨트롤러에 의해 자동으로 수행되기 때문이다. 백업과 리스토어를 그러나 NAND 플래시 메모리에서 DRAM으로의 데이터를 복원하거나, 전원 중단 없이 강제 백업을 위해서는 호스트 시스템에서 소프트웨어의 지원이 필요하다. 또한 NVDIMM-N 장치의 인식을 비롯한 활용을 위해서도 호스트 시스템 소프트웨어의 지원이 필요한데, 이에 대해서는 2.5 NVDIMM 소프트웨어에서 자세히 살펴본다.

이와 같이 NVDIMM-N은 DRAM과 동일한 성능을 갖는 비휘발성 메모리이면서 수명은 반영구적이고 호스트 시스템 하드웨어를 변경할 필요가 없기 때문에 점차 널리 사용되고 있다. 특히 DRAM과 동일한 성능을 제공하기 때문에 고성능이 필요한 응용, 특히 데이터베이스 시스템의 로깅 디바이스로 많이 활용되고 있다. 다른 NVDIMM 기술에 비해 구현이 간단하면서도 활용도가 높아, 가장 먼저 표준화가 진행되고 있으며 [19], 다수의 업체에서 NVDIMM-N 타입의 제품을 판매하고 있다. [20][21] [22] [23]. NVDIMM-N 장치는 데이터 백업과 복원을 담당할 컨트롤러가 필요하기 때문에, 주로 RDIMM 이나 LRDIMM 의 제어를 담당하는 RCD에 해당 기능을 추가하는 형태로 구현되고 있다.

2.4.2 NVDIMM-F

NVDIMM-F는 DRAM 인터페이스를 호스트 인터페이스로 활용하는 스토리지이며 본 연구에서 제안하는 스토리지는 DIMM 형태로 구현되지는 않았으나 NVDIMM-F에 해당된다. DRAM 인터페이스는 고정 레이턴시를 특징으로 하는 DRAM을 위한 인터페이스이므로, DRAM 이 아닌 NVM (주로 NAND 플래시 메모리)을 저장 매체로 사용하므로 가변 레이턴시를 갖는 NVDIMM-F를 DRAM 프로토콜 수준에서 직접 지원할수는 없다.

이 문제를 해결하기 위해서는 고정 레이턴시를 가정하는 DRAM 인터페이스에서 가변 레이턴시를 지원할 수 있도록 별도의 스토리지 인터페이스를 정의해야 한다. 따라서 스토리지 컨트롤러는 스토리지 인터페이스로 활용되는 공유 메모리 버퍼 또는 이와 유사한 형태의 메모리를 제공하고, 호스트 시스템은 이 메모리를 자신의 주소 공간에 매핑시켜서 사용해야 한다. 이것은 호스트의 DRAM 컨트롤러와 DRAM 장치가 완전한 마스터-슬레이브 구조이기 때문에 DRAM 인터페이스에서는 특정 DRAM (여기서는 스토리지 컨트롤러)이 다른 DRAM (호스트 시스템의 읽기/쓰기 명령에 대한 데이터가 저장된 버퍼)에 접근할 수 없기 때문이다. 따라서 호스트 시스템이 스토리지 컨트롤러가 제공하는 스토리지 인터페이스에 스토리지 명령과 데이터를 전달하면, 스토리지는 해당 인터페이스가 자신의 컨트롤러 내부에 존재하는 것이므로 호스트가 전달한 스토리지 명령과 데이터에 접근할 수 있다.

이처럼 스토리지 컨트롤러가 호스트 시스템이 사용할 스토리지 인터페이스를 제공하는 방식은 기존의 SATA, eMMC, UFS와 같이 호스트 시스템에 존재하던 별도의 스토리지 인터페이스 컨트롤러가 스토리지로 이

동한 것으로 간주할 수 있다. 이와 달리 PCIe를 기반으로 동작하는 NVMe의 경우는 마스터-슬레이브 구조가 아닌 포인트-투-포인트 구조이므로 스토리지 컨트롤러가 호스트 시스템의 메인 메모리에 접근할 수 있다. 따라서 명령 제출, 완료 큐, 명령에 대한 데이터가 포함된 버퍼는 호스트 시스템에 존재하고 스토리지 컨트롤러는 필요한 경우 호스트 시스템의 해당 영역에 접근함으로써 스토리지 명령에 대한 처리가 가능하다.

NVDIMM-F 인터페이스에서의 프로토콜은 스토리지 인터페이스에 해당되는 공유 메모리 버퍼를 어떤 방식으로 사용하는지에 대한 내용으로 간주할 수 있다. 프로토콜에 대해서는 아직까지 표준화가 진행되지 않았으며, 본 연구에서 제안하는 프로토콜은 3장에서 자세히 다룰 것이다.

NVDIMM-F의 장점은 DRAM 인터페이스에 기반하고 있기 때문에 빠른 전송속도와 매우 짧은 레이턴시를 제공할 수 있다는 것이다. 앞에서 살펴본 대로 DDR4 인터페이스의 경우 최대 32 GB/s 의 대역폭과 수 ns 수준의 레이턴시를 제공한다. DDR4 인터페이스의 대역폭은 PCIe 3.0 16레인의 최대 대역폭인 16 GB/s (단방향), 32 GB/s (양방향)과 비교시 2배 또는 동등한 수준이나, 레이턴시는 PCIe 대비 매우 우월하다. 따라서 DRAM 인터페이스를 스토리지 인터페이스로 활용하면 기존의 스토리지 인터페이스와 비교하여 우수한 전송속도와 레이턴시를 얻을 수 있다.

NVDIMM-F는 NVDIMM-N과 달리 표준화를 위한 작업이 매우 더디게 진행되고 있어 아직까지 프로토콜 및 에코시스템에 대한 표준화가 완료되지 않았다. 그러나 Diablo Technologies는 NVDIMM-F에 해당되는 유형의 제품을 *MCS (memory channel storage*라고 분류하고 [24], DDR3 인터페이스 기반으로 독자 규격을 이용해 상용화에 성공하였다 [25].

개발사에서 공개한 자료에 의하면 (그림 11), 앞에서 NVDIMM-F 인터페이스에 대해 기술한 것과 마찬가지로 스토리지 컨트롤러가 스토리지

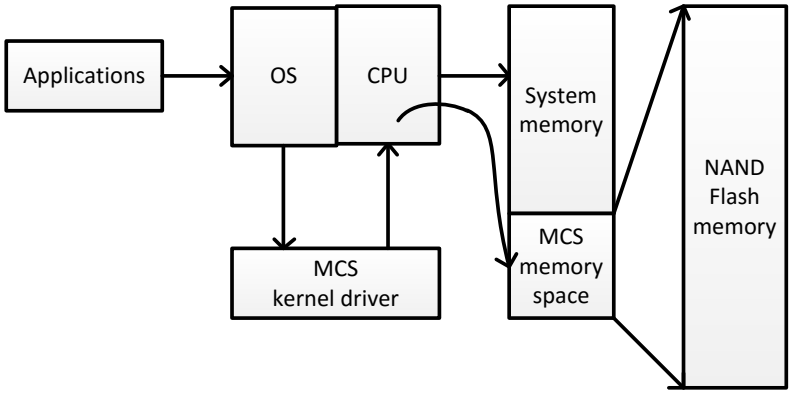

그림 11: ULLTraDIMM의 시스템 아키텍쳐

인터페이스용 메모리를 호스트 시스템에 제공하고, 호스트 시스템은 이 인터페이스용 메모리를 자신의 주소 공간에 매핑시켜서 스토리지 명령을 처리하는데 사용하는 것을 알 수 있다. 특히 해당 제품은 대용량 캐패시터를 사용하여 5 us 또는 3.3 us 수준의 쓰기 레이턴시를 제공하는 것이 특징이다. 이는 쓰기 명령과 데이터가 NVDIMM-F에 전달되면 DIMM에 전원 공급이 차단되어도 내장된 캐피시터를 활용하여 쓰기 작업을 끝까지 수행하여 영속성 (persistency)를 보장할 수 있기 때문에 스토리지가 쓰기 명령을 받더라도 실제 NAND 플래시 메모리에 쓰기 작업을 수행하지 않고 호스트에는 쓰기 명령이 완료된 것으로 보고하기 때문이다. 기존의 스토리지 인터페이스도 이와 같이 캐패시터를 활용하여 실제로 NAND 플래시 메모리에 쓰기 작업을 수행하지 않고도 쓰기 명령이 완료된 것으로 처리하는 것이 가능하나, PCIe 인터페이스 및 NVMe 프로토콜의 특성으로 인해 최저 10 us 수준인 것으로 알려져 있다.

본 연구와 기존의 상용화된 제품 모두 NVDIMM-F에 속하기 때문에

유사한 점이 있지만 다음과 같은 의의를 갖는다. 먼저 상용 제품의 내부 구조 및 동작 방식은 개념적으로만 일부 자료가 공개되어 있으나, 본 연구에서는 스토리지 내부와 호스트 시스템의 설계와 구현에 대해 자세한 내용을 기술한다. 또한 기존 제품은 기존의 SSD 컨트롤러와 DRAM 인터페이스를 위한 브릿지 컨트롤러를 결합한 형태로 구현되었으나, 본 연구에서는 SSD 컨트롤러와 DRAM 인터페이스를 통합한 단일 컨트롤러를 설계하고 구현하였다. 마지막으로 GPIO를 활용하여 명령 완료 처리에 대한 인터럽트를 발생시킴으로써 호스트 시스템이 명령 완료 처리를 위해 폴링을 해야 하는 필요성을 제거했다. 상용 제품의 명령 완료 처리 방식에 대한 자료가 공개되어 있지 않으므로 정확히 알 수는 없지만, DIMM 인터페이스의 특성상 에러 발생 등의 특별한 이벤트를 제외하면 인터럽트를 사용하지 않는 것으로 알려져 있으므로 아마도 폴링 방식으로 명령 완료 처리를 구현했을 것으로 추정된다.

NVDIMM-F는 DRAM 인터페이스를 기반으로 새로운 스토리지 프로토콜을 정의해서 사용하므로 하드웨어적인 변경은 불필요하며 소프트웨어 수준의 변경만 필요하다. NVDIMM-F는 본질적으로 SSD와 같이 블록 단위로 동작하는 스토리지이므로 운영체제의 디바이스 드라이버가 해당 장치를 블록 장치로 등록하면 기존의 SSD와 동일한 방식으로 사용할 수 있다. 따라서 어플리케이션을 전혀 수정하지 않아도 고성능 블록 디바이스를 활용할 수 있는 장점이 있다. 일반적으로 새로운 장치 또는 인터페이스를 사용하기 위해서는 신규 디바이스 드라이버를 개발과 신규 디바이스 드라이버를 적용해야 하는데, 일단 개발이 완료되면 비교적 간단한 절차를 통해 적용 가능하다. 어플리케이션을 수정해야 효과적으로 활용이 가능한 NVDIMM-N과 비교하면 기존의 어플리케이션을 그대로 사용 가능한 장점이 더욱 크게 부각된다.

그러나 신규 디바이스 드라이버 적용 뿐 아니라 BIOS (basic input output system)와 같은 시스템 초기화 소프트웨어의 변경 또한 필요한 것으로 알려져 있다. 변경이 필요한 이유는 NVDIMM-F를 일반 DRAM DIMM과 구분하여 인식하고 초기화 작업 등을 수행해야 하기 때문이며, 구체적으로 다음과 같은 절차가 필요하다 [25].

- MRC 변경

  MRC (memory reference code)는 x86 기반 컴퓨터 시스템의 BIOS에서 DRAM 컨트롤러를 초기화하고 시스템에 장착된 DRAM의 동작 타이밍등을 설정하는 역할을 담당한다 [26]. 이 과정에서 DRAM에 대해 간단한 테스트를 실시하고 인터리빙에 참여시키는데, NVDIMM-F는 DRAM이 아니기 때문에 메모리 테스트가 불필요하며 인터리빙에도 참여할 필요가 없다. 따라서 NVDIMM-F를 위한 바이오스는 MRC가 NVDIMM-F에 해당 작업을 수행하지 않도록 수정할 필요가 있다. 인터리빙과 관련된 내용은 6.3.2에서 자세히 다룬다.

- E820 테이블 변경

  E820은 x86 기반 컴퓨터 시스템의 바이오스가 운영체제에 메모리 맵을 전달하기 위한 테이블이다 [27]. 이 테이블에는 각 메모리 주소 영역이 사용 가능한지, 혹은 바이오스에 의해 예약되어 있는지 표시되어 있다. NVDIMM-F의 스토리지 인터페이스 영역은 운영체제가 일반적인 DRAM처럼 사용할 수 없는 영역이므로 예약 (reserved) 상태로 설정되어야 한다.

- MTRR 테이블 변경

  MTRR (memory type range registers)은 각 메모리 영역이 어떤 캐시

정책으로 관리되는지에 대한 정보를 제공하는 레지스터의 집합이다 [28]. 설정 가능한 캐시 모드는 *uncached*, *write-through*, *write-protect*, 그리고 *write-back*이 있다. NVDIMM-F의 스토리지 인터페이스 영역은 *uncached* 모드로 설정하여, 인터페이스 (메모리)에 쓰여진 데이터가 일단 CPU 내부의 캐시에 저장되도록 한다. 여기서 캐싱으로 인한 메모리 트랜잭션의 리오더링이 발생하여 쓰기 명령 처리시 무결성이 위반될 가능성이 있는데, 이는 6.3.4 메모리 트랜잭션 순서 보장에서 자세히 다룰 것이다.

- DIMM 트레이닝 절차 변경

  바이오스는 DRAM 인터페이스의 시그널에 대한 최적의 타이밍을 찾기 위한 조정 작업에 해당되는 트레이닝 과정을 수행한다 [29]. 이것은 하나의 시스템에 다수의 DRAM DIMM이 설치될 수 있으며, 메모리 컨트롤러부터 각 DRAM DIMM까지 시그널의 길이에 차이가 있기 때문에, 이에 대한 타이밍을 조절하기 위한 작업이다. 앞에서와 마찬가지로 NVDIMM-F의 스토리지 인터페이스 영역은 일반 DRAM이 아니므로 기존의 트레이닝 작업을 NVDIMM-F에 적합하도록 변경할 필요가 있다.

- ACPI 테이블 변경

  ACPI (advanced configuration and power interface)는 운영체제가 주변기기의 탐색, 설정, 전원관리를 위한 규약이다 [30]. 바이오스는 시스템 초기화 과정에서 주변기기에 대한 탐색 및 초기활르 수행한 후, 해당 주변기기에 대한 정보를 운영체제가 이용할 수 있도록 ACPI 규약에 맞춰 전달한다. 따라서 바이오스가 NVDIMM-F 에 대한 정보를 ACPI 테이블에 추가함으로써 운영체제가, 특히 디바이스

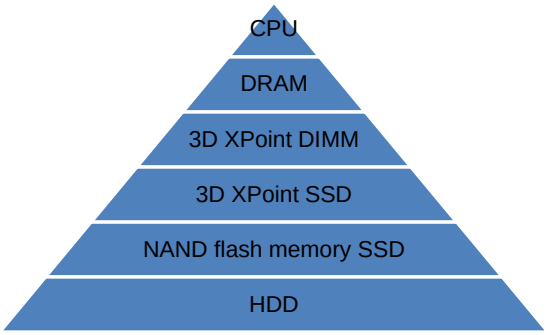

그림 12: 3D Xpoint가 반영된 메모리 계층 구조

드라이버가 이를 활용할 수 있도록 준비해야 한다.

2.4.3 NVDIMM-P

NVDIMM-P는 NVDIMM-F와 마찬가지로 개념만 정리되어 있는 상태이며 표준화가 완료되지 않았다. SNIA에서는 하나의 DIMM에서 NVDIMM-N과 NVDIMM-F의 기능을 모두 제공하는 것으로 정의하고 있다 [18]. 현재는 관련 표준도 존재하지 않고 상용화된 제품도 없는 상황이지만, 일부 업체가 NVDIMM-P의 개발에 착수한다는 발표가 있었다 [31].

2.4.4 Intel DIMM

Intel은 2015년에 비휘발성이면서 속도가 빠른 특성을 제공하는 3D Xpoint 메모리를 발표했다 [32]. 이 메모리 소자는 비휘발성이므로 스토리지에도 사용될 수 있으며, 읽기 속도가 DRAM에 근접하기 때문에 메인 메모리로도 사용할 수 있다. 기존의 메모리 계층 구조에 이 메모리 소자를 반영하면 그림 12의 계층 구조가 된다.

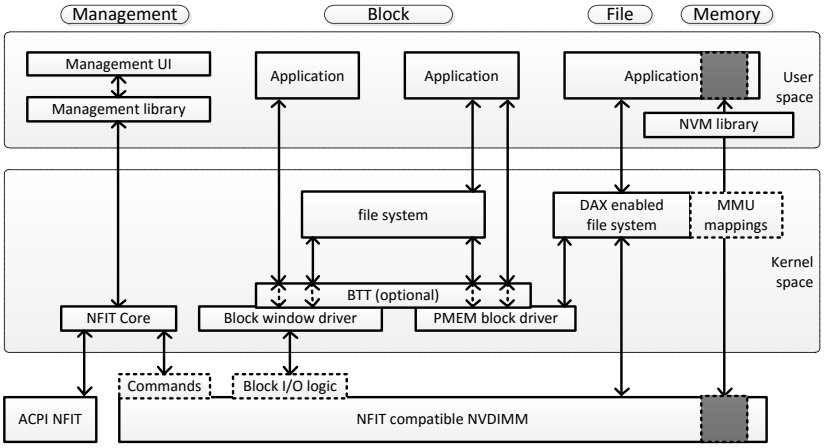

그림 13: NVDIMM 소프트웨어 아키텍쳐

2.5 NVDIMM 소프트웨어

NVDIMM은 기존의 DRAM DIMM과 다른 특성을 갖고 있기 때문에 효과적인 활용을 위해서는 시스템 소프트웨어의 지원이 필요하다. Intel 에서는 그림 13의 소프트웨어 아키텍쳐를 제안했다. 여기서 각 구성요소의 역할은 다음과 같이 정리할 수 있다. 먼저 시스템 초기화 소프트웨어가 NVDIMM을 DRAM DIMM과 구분하여 인식하고, 이에 대한 정보를 운영체제에 전달해야 한다. 운영체제는 전달받은 정보를 이용해서 해당 NVDIMM을 관리해야 하는데, 이를 위해서는 디바이스 드라이버가 반드시 필요하다. 또한 해당 장치의 특성을 활용하기 위해 가상 메모리 관리자나 파일 시스템과 같은 운영체제의 구성요소들을 수정해야 할 수도 있다. 마지막으로 사용자 어플리케이션이 NVDIMM 장치를 활용하기 위한 API와 라이브러리, 그리고 이를 이용한 사용자 어플리케이션의 수정이 필요하다. 이제부터는 NVDIMM 소프트웨어의 역할에 대해 상세히 살펴보기

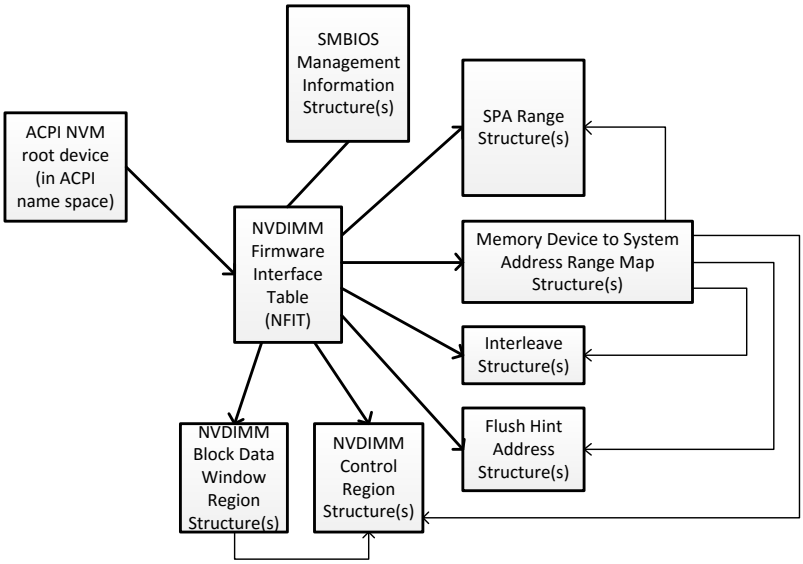

그림 14: NVDIMM firmware interface table

로 하자.

2.5.1 BIOS

시스템 초기화를 담당하는 바이오스는 DRAM 서브시브템 초기화시에 NVDIMM을 일반 DRAM DIMM과 구분하여 초기화하고, 관련 정보를 운영체제에 전달해야 한다. ACPI 6.0 에서는 이를 위해 NFIT (NVDIMM firmware interface talbe)를 정의했다 (그림 14) [33]. NFIT는 운영체제가 시스템에 존재하는 NVDIMM을 확인하고 NVDIMM이 제공하는 시스템의 물리 주소 영역과 연관 관계를 형성할 수 있는 정보를 제공하는 역할을 한다. NFIT의 구성 요소와 각각의 역할은 다음과 같다.

- SPA Range Structure

NVDIMM이 사용하는 시스템의 물리 주소 영역과 각 영역의 유형을 기술한다.

- Memory Device to System Address Range Map Structure

  메모리 장치의 물리적인 위치, 시스템 물리 주소 (System Physical Address, SPA), NVDIMM 영역, 메모리 인터리브 설정, 다바이스 상태 플래그를 기술한다.

- Interleave Structure

  현재 플랫폼에서 활성화된 다양한 인터리브 옵션을 기술한다. *Memory Device to System Address Range Map Structure*는 현재의 메모리 인터리브 정보를 기술하고 있는 Interleave Structure를 포인트한다.

- SMBIOS Management Information Structure

  SMBIOS 테이블 항목을 기술한다. 이것은 새로 추가된 NVDIMM에 대한 SMBIOS 엔트리를 기술할 수 있도록 한다.

- NVDIMM Control Region Structure

  NVDIMM을 기술하고 가능한 경우는 Block Control Window 까지 기술한다.

- NVDIMM Block Data Window Region Structure

  Block 동작이 가능한 NVDIMM에 대한 Block Data Window를 기술한다.

- Flush Hint Address Structure

  NVDIMM Region에 대한 write에 대해 영속성 (durability)를 확보하는데 도움을 줄 수 있는 특별한 시스템 물리 주소를 기술한다. Flush

Hint Address 지원은 플랫폼에 한정적이며, 각 구조체 항목은 NFIT Device Handle을 통해 구조체에 의해 지원되는 NVDIMM을 기술한다.

2.5.2 Linux 지원

현재 리눅스에는 그림 13의 NVDIMM을 위한 구성 요소 대부분이 지원되고 있으며 각 구성요소별 역할은 다음과 같다.

- NFIT Core

  2.5.1 BIOS에서 기술한 NFIT 인터페이스 정보를 이용해서 시스템에 장착된 NVDIMM 장치를 관리하는 모듈이다.

- Block Window Driver

  NFIT는 NVDIMM 장치에서 블록 단위의 I/O를 할 수 있는 Block Window를 정의했다. Command/Address Register를 통해 명령을 전송하고 Status Register를 통해 결과를 확인한다. 이 모듈은 해당 인터페이스를 이용한 블록 I/O를 수행할 수 있도록 한다.

- PMEM Block Driver

  바이트 단위로 접근할 수 있는 NVDIMM 장치를 블록 디바이스로 사용할 수 있는 기능을 제공한다.

- BTT

  BTT (Block Translation Table)는 커널 내부의 논리-물리 매핑 정보를 유지함으로써 atomic write 기능을 제공한다.

- NVM Library

2.5.4 NVM Programming Model에서 제안한 NVM Programming Model의 구현이다.

파일 시스템의 경우는 NVDIMM-N 또는 SCM을 위한 ext4, BPFS, PMFS 파일 시스템을 사용할 수 있다. NVDIMM-N과 SCM 모두 공통적으로 바이트 단위의 접근과 비휘발성을 특징으로 하므로 시스템 소프트웨어의 많은 부분을 공유할 수 있다.

기존의 파일시스템과 다른 점은 DAX (direct access) 기능을 활용한다는 점이다. DAX 기능을 사용하면 페이지 캐시를 사용하던 기존의 파일시스템과 다르게 페이지 캐시를 사용하지 않고 NVDIMM 장치에 직접 접근해서 파일 시스템 작업을 수행한다. 따라서 파일 시스템의 성능을 크게 개선할 수 있다.

2.5.3 Microsoft Windows 지원

Microsoft Windows 운영체제도 NVDIMM과 SCM을 위한 소프트웨어 개발을 진행하고 있다. 현재 가장 먼저 상품화가 진행된 NVDIMM-N을 활용하기 위한 인터페이스 [34]가 정의되어 있다. 또한 SCM driver 개발이나 기존 파일시스템을 NVDIMM-N과 SCM에 적합하도록 수정하는 작업이 진행되고 있다 [35].

2.5.4 NVM Programming Model

SNIA (Storage Networking Industry Association)는 NVDIMM, SCM 등의 NVM을 위한 프로그래밍 모델을인 NVM Programming Model을 제안했다. 여기서는 다음과 같이 4가지의 모델을 제안하고 있다.

1. NVM.BLOCK

NVM.BLOCK 모드는 블록 장치로 동작하는 NVM 장치를 위한 프로그래밍 인터페이스를 제공한다. 이 프로그래밍 인터페이스는 I/O 명령을 블록 드라이버로 전송하는 운영체제의 원래 동작에 NVM을 위한 확장을 추가한다. 이 모드를 지원하기 위해서는, NVM 장치가 NVM에 대한 명령 인터페이스를 제공하는 NVM block capable driver에 의해 지원되어야 한다.

추가된 주요 인터페이스는 다음과 같다.

- NVM.COMMON.GET_ATTRIBUTE
- NVM.COMMON.SET_ATTRIBUTE
- NVM.BLOCK.ATOMIC_WRITE
- NVM.BLOCK.ATOMIC_MULTIWRITE
- NVM.BLOCK.DISCARD_IF_YOU_CAN
- NVM.BLOCK.DISCARD_IMMEDIATELY
- NVM.BLOCK.EXISTS
- NVM.BLOCK.SCAR

2. NVM.FILE

NVM.FILE 모드는 NVM 장치에 대한 파일 I/O 인터페이스를 제공한다. 추가된 주요 인터페이스는 다음과 같다.

- NVM.COMMON.GET_ATTRIBUTE
- NVM.COMMON.SET_ATTRIBUTE
- NVM.FILE.ATOMIC_WRITE

- NVM.FILE.ATOMIC_MULTIWRITE

3. NVM.PM.VOLUME

   NVM.PM.VOLUME 모드는 persistent 메모리 하드웨어에 대한 PM volume 이라는 추상화 계층을 제공한다. PM 볼륨은 메모리 매핑 기능을 지원함으로써 기존의 CPU에서 사용하는 load, store 인스트럭션을 그대로 사용할 수 있다. PM 볼륨은 메모리 채널, PCIe 또는 어떠한 다른 방법을 통해서도 제공될 수 있다. PM 볼륨에 접근하기 위해서 운영체제가 컨텍스트 스위치를 해야 한다는 요구사항이 있어서는 안된다. 추가된 주요 인터페이스는 다음과 같다.

   - NVM.COMMON.GET_ATTRIBUTE
   - NVM.COMMON.SET_ATTRIBUTE
   - NVM.PM.VOLUME.GET_RANGESET
   - NVM.PM.VOLUME.VIRTUAL_ADDRESS_SYNC
   - NVM.PM.VOLUME.DISCARD_IF_YOU_CAN
   - NVM.PM.VOLUME.DISCARD_IMMEDIATELY
   - NVM.PM.VOLUME.EXISTS

4. NVM.PM.FILE

   NVM.PM.FILE 모드는 사용자 어프리케이션이 NVM을 메모리로서 직접 접근할 수 있는 수단을 제공한다. 이 모드의 많은 동작은 기존의 파일시스템이 제공하는 API로 구현되도록 의도되었다. NVM.PM.FILE 구현은 기존의 파일 시스템 구현과 유사하나, 사소한 차이가 있다.

   추가된 주요 인터페이스는 다음과 같다.

- NVM.COMMON.GET_ATTRIBUTE
- NVM.COMMON.SET_ATTRIBUTE
- NVM.PM.FILE.MAP
- NVM.PM.FILE.SYNC
- NVM.PM.FILE.OPTIMIZED_FLUSH
- NVM.PM.FILE.GET_ERROR_EVENT_INFO
- NVM.PM.FILE.OPTIMIZED_FLUSH_AND_VERIFY

제 3 장

스토리지 인터페이스 설계

이 장에서는 본 논문에서 제안하는 스토리지 인터페이스를 논의한다. 먼저 호스트 시스템과 스토리지의 물리 계층 수준의 연결 방식을 제안하고, 제안된 물리 계층상에서 소프트웨어 수준의 프로토콜을 정의한다. 마지막으로 프로토콜 설계 시 고려해야 할 사항을 논의한다.

3.1 물리 계층

본 논문에서 제안하는 전체 시스템 아키텍쳐는 그림 15와 같다. 제안된 스토리지 장치는 DRAM 버스와 인터럽트 시그널을 이용해서 호스트 시스템의 CPU와 통신한다. 스토리지 장치와 호스트 시스템 사이의 데이터 이동은 호스트 소프트웨어가 처리한다.

스토리지 장치의 호스트 인터페이스는 마치 DRAM 처럼 동작한다. 따라서 호스트 시스템은 제안된 스토리지 장치를 사용하기 위해서 호스

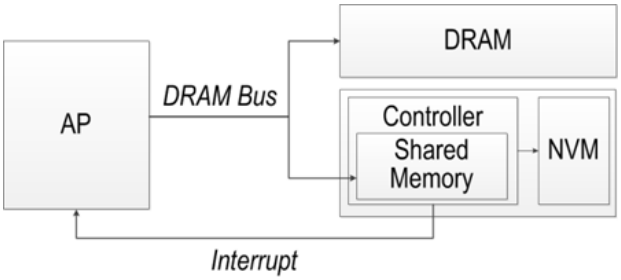

그림 15: 전체 시스템 아키텍쳐

트 시스템을 변경할 필요가 없다. 이를 위해서 작은 크기의 공유 메모리 버퍼를 호스트 인터페이스로 사용한다. 이 버퍼는 호스트 시스템 CPU와 스토리지 컨트롤러에게 공유된다. 호스트 시스템 CPU는 해당 버퍼를 일반적인 DRAM과 마찬가지로 일반적인 *load*와 *store* 명령을 통해 접근할 수 있다. 호스트 시스템 CPU와 스토리지 컨트롤러가 이 버퍼에 동시에 접근하는 경우에는, 해당 공유 버퍼의 컨트롤러가 호스트 시스템 CPU에 높은 우선 순위를 준다. 이러한 접근 우선순위 제어를 통해서 호스트 시스템에게 공유 메모리 버퍼의 시간적인 동작 특성을 일반적인 DRAM 과 동일하게 제공할 수 있다.

공유 메모리 버퍼는 그 자체로는 단순한 메모리이며, 메모리의 내용은 하드웨어에 의해 해석되지 않는다. 그 대신, 몇 개의 특수 기능 레지스터들이 메모리의 특정 영역에 매핑되어 있다. 특수 기능 레지스터들은 호스트 소프트웨어가 스토리지 장치에 특별한 제어 정보를 전달하거나, 혹은 반대로 스토리지 장치가 호스트에 특별한 정보를 전달하는데 사용된다. 이와 같은 물리적인 기법들은 DRAM 인터페이스에 스토리지 장치 프로토콜을 내재시키는데 충분한 유연성을 제공한다.

3.2 스토리지 프로토콜

여기서는 앞에서 정의한 물리 계층 위에서 동작하는 스토리지 장치 프로토콜을 논의한다. 프로토콜의 전반적인 동작은 그림 16에 정리되어 있다. 스토리지 장치 인터페이스의 핵심인 공유 메모리 버퍼는 다음의 세 가지 영역으로 나눠진다: *제출 큐 (SQ: Submission Queue)*, *완료 큐 (CQ: Completion Queue)*, 그리고 *데이터 버퍼 (DB: Data Buffer)*. 호스트는 제출 큐를 이용해 스토리지 장치에 명령을 전달하고, 완료 큐를 이용해 명령 완

료 상태를 확인한다. 읽기 또는 쓰기 명령을 위한 데이터는 데이터 버퍼를 통해서 교환한다.

하나의 제출 큐 엔트리는 다음과 같은 항목으로 구성된다: *명령 ID (command ID)*, *연산 코드 (opcode)*, *시작 섹터 (start sector)*, *섹터 개수 (sector count)*, 그리고 *버퍼 인덱스 (buffer index)*. 명령 ID는 동시에 실행 중인 여러 개의 명령 중에서 특정 명령을 식별하기 위해 사용된다. 연산 코드는 실행할 명령의 종류를 나타내며 읽기, 쓰기 등이 있다. 시작 섹터와 섹터 개수는 명령의 대상이 되는 시작 섹터와 섹터 개수이다. 버퍼 인덱스는 데이터 버퍼 내부의 인덱스이며, 버퍼 주소와 유사하다. 읽기 또는 쓰기 명령에 대한 데이터는 이 버퍼 인덱스로부터 시작해서 연속적으로 저장된다.

하나의 완료 큐 엔트리는 *명령 ID (command ID)*와 *완료 상태 (completion status)*의 2개 항목으로 구성된다. 명령 ID는 실행이 완료된 명령의 명령 ID인데, 이 값은 해당 명령이 제출 큐에 제출될 때 주어진 값을 사용한다. 완료 상태는 이 명령이 성공적으로 완료되었는지 아닌지를 나타낸다.

제안된 프로토콜이 동작하는 예를 그림 16과 함께 살펴보자. 먼저 호스트 시스템이 수행하려는 스토리지 명령을 제출 큐 엔트리 양식에 맞춰 준비한 후 제출 큐의 끝 (tail)에 저장한다 (#2). 제출한 명령이 쓰기 명령인 경우, 명령을 제출하기 전에 쓰기 데이터를 데이터 버퍼에 저장해두어야 한다 (#1). 호스트 시스템은 새로운 명령이 제출되었음을 스토리지에게 알려주기 위해서 갱신된 현재의 제출 큐의 끝 인덱스를 NVMe 스펙 [3]의 도어벨과 유사한 레지스터에 기록할 수도 있다.

이제 스토리지 장치는 제출 큐에 새로 제출된 명령이 있는지 확인하고, 만약 있다면 새로 제출된 명령을 가져온다 (#3). 신규 제출된 명령의

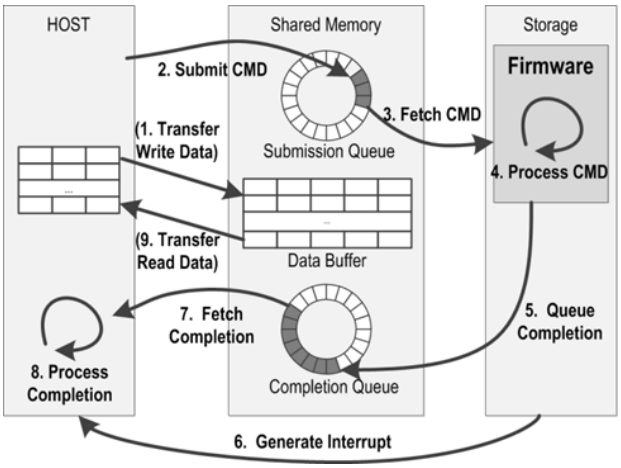

그림 16: 스토리지 프로토콜 실행 순서

탐색은 제출 큐의 선두 (head)부터 시작해서 끝 (tail) 의 순서로 진행한다. 스토리지 장치는 제출 큐에서 가져온 명령을 실행한다 (#4). 실행할 명령이 쓰기 명령인 경우 스토리지에 저장할 데이터는 데이터 버퍼에 저장되어 있다. 해당 데이터의 데이터 버퍼상의 시작 위치는 제출 큐 엔트리의 버퍼 인덱스 항목에 지정되어 있다. 실행할 명령이 읽기 명령인 경우 스토리지 내부의 읽기 명령 실행이 완료되면, 제출 큐 엔트리에서 지정한 데이터 버퍼 인덱스에 데이터 버퍼상의 위치에 읽기 명령에 해당되는 데이터를 저장해야 한다.

스토리지 장치 내부에서 명령 실행이 완료되면 완료 큐의 끝 (tail)에 완료 엔트리를 저장하고 (#5), 호스트 시스템에게 명령이 실행이 완료되었음을 알린다 (#6). 호스트 시스템은 새로 도착한 완료 엔트리를 완료 큐의 선두 (head) 부터 끝 (tail) 의 순서로 검색한다 (#7). 새로 도착한 완료 엔트리가 있으면 호스트 시스템은 해당 엔트리를 처리한다 (#8). 완료된 명령이 읽기 명령이라면, 데이터 버퍼에 저장되어 있는 스토리지의 읽기

명령에 대한 데이터를 호스트 시스템의 버퍼로 복사한다 (#9).

프로토콜 실행 흐름에서, 실제 발생하는 이벤트들은 호스트 시스템에서 실행되는 디바이스 드라이버가 요청한 메모리 접근 (호스트 시스템에서 스토리지 장치 방향)과 인터럽트 신호 생성 (스토리지 장치에서 호스트 시스템 방향)으로 구성된다. 호스트 시스템의 버퍼와 공유 메모리 버퍼 사이의 데이터 전송 (#1 과 #9)은 호스트 시스템에 의해 수행된다. 공유 메모리 버퍼와 스토리지 매체 사이의 데이터 전송은 스토리지 컨트롤러의 DMA 엔진이 담당한다.

마지막으로, 본 프로토콜과 같이 두 개체 사이의 통신을 위해 큐를 사용하는 것은 스토리지 및 통신 시스템에서는 흔한 설계 패턴이다. 큐는 한 시스템 내에서 동시에 여러 개의 메시지 (스토리지 명령과 완료 결과)가 처리될 수 있도록 한다. 이 특성은 병렬성과 접근 시간의 변동성이 존재하는 스토리지 장치에서는 매우 중요하다. 다수의 NAND 플래시 메모리 칩으로 구성된 스토리지 장치는 일반적으로 칩의 개수만큼 병렬성을 제공하며, 접근 시간은 현재 스토리지 장치의 상태와 입력으로 주어진 명령에 의해 변동되기 때문이다. 그림 16에는 각각 1개의 제출 큐와 완료 큐가 포함되어 있으나, 필요에 따라 큐의 개수를 증가시킬 수 있다.

3.3 고려사항

3.3.1 명령 실행 완료 전달

전통적인 스토리지 장치 인터페이스는 명령 실행 완료 통보를 위한 장치들을 포함하고 있다. 예를 들면, 호스트 시스템의 eMMC 호스트 컨트롤러는 그 자신과 eMMC 장치의 컨트롤러간의 상호 동작을 숨기고 있다가 명령이 완료되면 명령 실행 완료에 대한 인터럽트를 생성한다. 또 다른

예로는, NVMe SSD는 기반으로 사용하고 있는 PCIe 버스의 인터럽트 기법 (기존의 시그널링 방식 또는 메시지 기반)을 활용한다. 그러나 DRAM 인터페이스는 호스트 시스템에 명령 실행이 완료되었음을 전달하기 위한 장치를 제공하지 않는다. DRAM이 결정적인(deterministic) 동작 시간을 갖는 수동적인 소자이기 때문이어서 비동기적인 사건 발생을 전달하기 위한 장치를 필요로 하지 않기 때문이다.

그러나 스토리지 장치의 명령 실행시간은 비결정적 (non-deterministic) 이므로 비결정적인 실행 시간에 대응할 수 있는 방법이 필요하다. 이에 대한 첫 번째 해결 방법은 호스트 시스템이 폴링 (polling) 하는 것이다. 조금 더 구체적으로는, 호스트 시스템의 디바이스 드라이버가 제출 큐에 명령을 제출한 후, 주기적으로 완료 큐를 확인하면서 제출된 명령의 실행 완료 여부와 완료 상태를 점검할 수 있다. 또 다른 해결 방법은 호스트 시스템에서 이미 제공하고 있는 통보 기법을 사용하는 것이다. 일반적인 모바일 시스템의 경우, 모바일 CPU는 GPIO 핀을 가지고 있는데, 이 핀은 외부의 인터럽트 신호를 수신할 수 있도록 설정 가능하다. 호스트 시스템의 CPU 자원을 절약하기 위해서 본 연구에서는 두 번째 방법, 즉 GPIO 핀을 이용한 명령 실행 완료 전달 방법을 사용한다. 그러나 매우 높은 I/O 성능을 제공하는 경우에는 폴링이 인터럽트보다 효과적일수도 있다 [36]. 폴링과 인터럽트 방식에 의한 이벤트 처리는 전통적인 연구 주제이며 각각의 장단점을 갖고 있으나 본 연구에서는 이에 대해 자세히 다루지 않을 것이다.

이 문제는 본 연구 뿐 아니라 고성능 SCM과도 관련된다. 현재는 SCM을 컴퓨터 시스템에 연결하는 표준화된 인터페이스가 존재하지 않는다. 아직까지는 SCM이 연구 단계여서 독자적인 인터페이스를 사용하거나 시뮬레이션을 통해 평가를 진행하기 때문에 큰 문제가 되지 않는다. 그러나 SCM이 실제로 활용할만한 수준이 된다면 현재 컴퓨터 시스템에

서 사용할 수 있는 인터페이스가 필요한데, 본 연구와 같은 이유로 DRAM 인터페이스를 그대로 사용할수는 없다. SCM이 DRAM과 동일한 타이밍 특성을 갖는다면 그대로 사용할 수 있겠지만 가능성이 매우 적다. 따라서 기존의 DRAM 프로토콜을 확장하거나 새 프로토콜을 제안하여 다양한 SCM을 수용할 수 있는 준비가 필요하다.

3.3.2 명령 제출시 무결성 보장

그림 16의 프로토콜 실행 순서에 대한 정확한 처리는 메모리 일관성 (consistency) 모델에 의존한다. 그림 16의 #1과 #2를 예로 들어 살펴보자. 호스트 시스템은 쓰기 명령이 제출되기 전에 쓰기 데이터가 데이터 버퍼에 저장되어 있음을 보장해야 한다. 만약 이 가정이 지켜지지 않으면 스토리지 컨트롤러는 데이터 버퍼에 있는 불완전한 데이터를 스토리지 매체에 저장할 수도 있다. 따라서 호스트 시스템은 프로토콜의 두 과정, 즉 쓰기 명령 제출과 데이터 버퍼로 쓰기 데이터를 복사하는 프로토콜 간의 순서를 보장할 수 있도록 적절한 기법을 활용해야 한다.

본 연구에서 사용한 호스트 플랫폼에서는 CPU의 L1, L2 캐시로 인한 문제는 발생하지 않았다. 호스트 시스템의 디바이스 드라이버가 명령 제출시의 무결성을 보장하도록 설계되어야 하는데, 이를 위해 공유 메모리 버퍼 공간을 캐시가 불가능한 영역으로 간주하고 적합한 메모리 작업 순서를 적용해야 한다. 이 주제는 NVDIMM과 SCM 모두와 관련이 있으며 자세한 내용에 대해서는 6.3.4 메모리 트랜잭션 순서 보장에서 논의한다.

3.3.3 데이터 전송 부하 분배

호스트 시스템에 별도의 스토리지 인터페이스를 위한 컨트롤러를 포함하고 있는 기존의 스토리지 인터페이스와 달리, 본 스토리지 인터페이스는 호스트 시스템의 DRAM 컨트롤러 이외에는 별도의 컨트롤 로직을 포함하지 않는다. 따라서 호스트 시스템의 소프트웨어가 스토리지 인터페이스에 해당되는 공유 메모리 버퍼와 호스트 시스템의 버퍼 사이의 데이터 전송을 관리해야 한다.

데이터 전송에는 CPU 자체의 인스트럭션에 기반한 메모리 복사를 이용할 수 있다. 가장 기초적인 방법으로는 *load*, *store* 인스트럭션을 반복함으로써 복사하는 방법이 있는데 어느 호스트 시스템에서나 이용가능한 방법이다. 그러나 데이터 복사에 CPU가 계속해서 개입해야 하므로 단순 반복적인 작업에 귀중한 CPU 자원을 낭비하게 됨으로써 전체적인 시스템 성능이 저하될 가능성이 있다 [37]. 또다른 문제로는 쓰기 데이터의 경우 공유 메모리 버퍼에 복사된 후 스토리지에 저장되면 당분간 접근될 확률이 낮은 데이터임에도 불구하고 해당 데이터가 L1, L2 의 다른 데이터를 밀어내고 자신이 남아있는 캐시 오염 문제이다. 따라서 CPU를 이용한 데이터 복사는 이용하지 않는 것이 좋다. 특정 CPU 아키텍쳐에서는 대량의 데이터 복사에 최적화된 별도의 연산을 제공하는 경우도 있으나 [38], 해당 CPU 아키텍쳐에서만 사용 가능하다는 단점이 있다.

대량의 데이터 복사에 적합한 방법은 전통적으로 사용되던 DMA (Direct Memory Access) 장치를 이용한 방법이다. 본 연구에서도 DMA 장치를 활용함으로써 호스트 시스템의 버퍼와 스토리지 인터페이스 버퍼 사이의 데이터 전송을 효과적으로 수행했다. 일반적으로 최신 모바일 시스템 CPU 에는 범용 DMA 장치가 포함되어 있고 [39], 이것을 활용하

면 데이터의 전송 시작과 종료시에만 CPU가 개입하면 된다. 따라서 낮은 CPU 오버헤드만으로도 대용량의 데이터를 고속으로 전송할 수 있다. 하지만 데이터의 양이 작은 경우는 DMA 장치를 동작시키는 비용이 더 클 수 있으므로 CPU 인스트럭션을 이용한 복사가 적합할 수 있다. 이에 대한 자세한 설계는 5장에서 다룬다.

제 4 장

스토리지 설계

이 장에서는 3장에서 제안한 스토리지 인터페이스를 지원하는 스토리지를 설계한다. 먼저 스토리지 인터페이스, NAND 플래시 메모리 제어, 펌웨어 실행을 위한 리소스 등을 제공하는 컨트롤러를 설계한다. 그 후 해당 스토리지 컨트롤러를 기반으로 NAND 플래시 메모리를 섹터 단위로 접근 가능한 스토리지로 동작시키는 펌웨어를 설계한다.

4.1 컨트롤러

본 연구에서 제안하는 스토리지 컨트롤러의 전체 구조는 그림 17과 같다. 컨트롤러는 세 블록으로 구성된다: *호스트 인터페이스*, *백본 (backbone)*, *플래시 메모리 인터페이스*. 이 블록들은 서로 AXI 인터커넥트로 연결되어 있어서 고속으로 데이터를 교환할 수 있다. 각 블록의 기능은 다음과 같다.

- 호스트 인터페이스 블록

  본 연구에서 제안한 인터페이스의 핵심인 공유 메모리 버퍼를 제공하며, 이 버퍼는 호스트 시스템 관점에서 DRAM 처럼 동작한다. 호스트 시스템과 스토리지 장치는 이 공유 메모리 버퍼를 이용해서 스토리지 명령, 명령의 처리 결과, 읽기/쓰기 명령의 데이터를 교환한다.

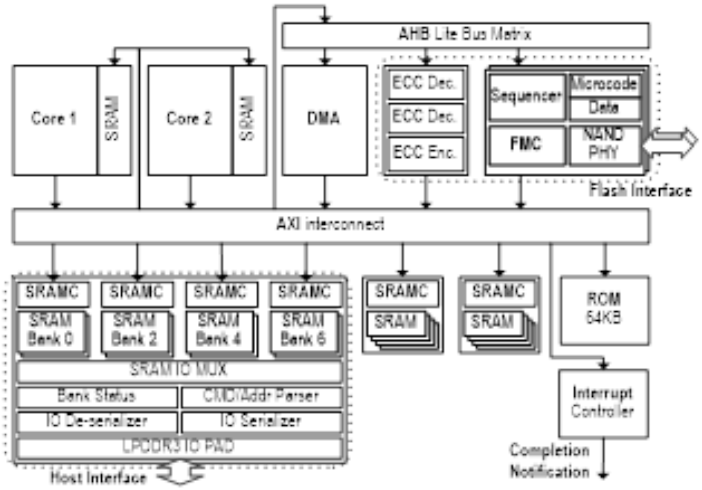

그림 17: 스토리지 컨트롤러 설계

- 백본 블록

  펌웨어 동작에 필요한 기능을 제공한다. CPU, 메모리와 대용량 데이터 이동에 필요한 DMA 하드웨어 등을 제공함으로써 펌웨어가 원활히 동작할 수 있도록 한다.

- 플래시 메모리 인터페이스 블록

  다수의 NAND 플래시 메모리 컨트롤러와 ECC 엔진을 제공한다. 이를 통해 고성능, 고신뢰성을 갖는 NAND 플래시 메모리 연산을 수행할 수 있도록 한다.

이제는 각 블록의 설계에 대해 상세히 살펴본다.

4.1.1 호스트 인터페이스 블록

이 블록은 본 연구에서 제안한 호스트 인터페이스의 핵심인 공유 메모리 버퍼를 제공한다. 이 공유 메모리 버퍼는 에뮬레이트된 DRAM 장

치이므로, 호스트 시스템은 이 공유 메모리 버퍼를 일반적인 DRAM과 동일하게 사용할 수 있다.

이 공유 메모리 버퍼는 여러개의 SRAM 블록과 관련 로직으로 구성된다. 본 연구에서는 이 버퍼가 LPDDR3 스펙과 호환되도록 설계했으며, 이를 위해 기존의 DRAM 장치와 동일한 I/O PAD를 사용한다. I/O PAD의 뒤쪽에는 명령/주소 해석기 (command/address parser)가 있어서 I/O PAD에 도달한 DRAM 명령과 주소를 해석하여 명령과 대상 주소를 결정한다. 대상 주소는 각 뱅크 (bank)의 row 주소를 유지하고 있는 뱅크 상태 레지스터 (bank status register)로 전달되어 각 요청에 대해 row와 column 주소가 연결된 (concatenated) 주소를 생성한다. 이 버퍼는 8개의 SRAM 뱅크로 구성되는데, 각 뱅크는 여러 개의 SRAM 매크로로 구성된다. 각 SRAM 매크로는 (듀얼 포트가 아닌) 싱글 포트를 사용하여 SRAM 로직 영역을 최소화한다.

3.1 물리 계층에서 논의된 대로, 이 버퍼는 호스트 시스템과 스토리지 컨트롤러가 공유하기 때문에 양 쪽에서 모두 접근 가능하다. 만약 이 버퍼에 호스트와 스토리지가 동시에 접근하는 경우는 충돌이 발생한다. 이렇게 충돌이 발생하는 상황에서는 호스트 시스템이 먼저 접근할 수 있도록 우선권을 준다. 호스트 시스템에게 우선권을 주는 이유는 다음과 같다. 이 버퍼는 호스트 시스템이 DRAM 장치로 간주하고 있기 때문에 DRAM 장치와 동일한 동작 시간 특성을 제공하도록 에뮬레이션 해야 한다. 만약 충돌이 발생했을 경우 스토리지 컨트롤러의 접근에 우선권을 준다면 호스트 시스템이 기대하는 DRAM 장치의 동작 시간 특성을 충족시킬 수 없다. 따라서 공유 버퍼는 호스트 시스템의 접근을 먼저 처리해야 한다.

단, 버퍼로의 동시 접근에 의한 충돌 발생 시 스토리지 컨트롤러의 접근이 차단되므로 약간의 성능 저하가 발생할 수 있다. 충돌로 인한 성능

저하를 최소화하기 위해 그림 17과 같이 8개의 SRAM 뱅크 각각에 대해 별도의 SRAM 컨트롤러를 사용한다. 이 경우 호스트 시스템과 스토리지 컨트롤러가 같은 뱅크에 접근하지 않는 한 충돌이 발생하지 않는다. 따라서 단일 SRAM 컨트롤러를 사용할 때와 비교해 충돌 발생 확률이 1/8 로 감소한다.

4.1.2 백본 (backbone) 블록

이 블록은 스토리지를 구동시키는 펌웨어를 위한 기본적인 장치들, 즉, CPU, SRAM, DMA 등을 제공한다. 여기에는 두 개의 동일한 CPU 가 포함되는데, 명령어와 데이터를 저장하기 위해 전용 스크래치패드 메모리를 사용한다. 스크래치패드 메모리 덕분에 각 CPU는 캐시 서브시스템을 사용하지 않고도 안정적로 높은 성능을 제공한다.

또한 범용으로 사용되는 두 개의 SRAM 모듈이 포함되는데, 각 모듈은 독립적인 SRAM 컨트롤러에 의해 제어됨으로써 높은 성능을 낼 수 있다. AXI 인터커넥트의 메모리 매핑 로직은 SRAM에 대한 접근을 인터리브 방식으로 처리하기 대문에 충돌 발생 가능성을 감소시킨다. 백본 블록에 포함된 범용 DMA는 소프트웨어가 대량의 데이터를 효과적으로 전송할 수 있도록 한다. DMA에는 데이터 전송 기능 뿐만 아니라 AND, XOR 등의 비트 연산 기능이 포함되어 있다. 이 기능을 이용하면 데이터 인코딩이나 디코딩과 같은 시간이 오래 걸리는 데이터 처리 작업을 가속화하여 처리할 수 있다.

이 블록은 인터럽트 컨트롤러, 타이머, GPIO 등의 주변장치를 위한 IP 들도 포함한다. 인터럽트 컨트롤러는 호스트에 스토리지 명령의 완료를 통보하는데 중요한 역할을 담당한다. 이 컨트롤러는 하나의 스토리지 명령이 완료될 때 마다 인터럽트 신호를 생성함으로써, 호스트 시스템이

명령 완료를 확인하기 위한 폴링의 필요성을 제거한다.

4.1.3 플래시 메모리 인터페이스 블록

이 블록은 NAND 플래시 메모리 칩을 제어하기 위한 장치들을 제공하며, 시퀀서 (sequencer)와 플래시 메모리 컨트롤러 (FMC, Flash Memory Controller) 등이 포함된다. 시퀀서는 특수 목적의 마이크로 컨트롤러인데, 고수준의 제어 신호 시퀀스를 생성한다. 플래시 메모리 컨트롤러는 시퀀서가 생성한 신호 시퀀스를 받아서 이것을 NAND 플래시 메모리 칩을 제어하기 위한 저수준의 제어 신호와 데이터 신호를 생성한다. 또한 플래시 메모리 컨트롤러는 내부의 SRAM과 NAND 플래시 메모리 사이에서 사용자 데이터를 전송한다. NAND 플래시 메모리 PHY는 하드 매크로로 구현되는데, 플래시 메모리 컨트롤러와 NAND 플래시 메모리 칩 사이에서 데이터를 고속으로 전송한다. 이것은 유효한 데이터 윈도우의 중앙을 탐색하고 중앙 정렬된 데이터 스트로브를 생성하기 위해서 delay-locked loops (DLLs)와 지연 제어 로직을 사용한다.

마지막으로, 이 블록은 두 개의 공유된 ECC 디코더와 한 개의 ECC 인코더를 포함한다. 이렇게 비대칭적인 구조를 통해 본 연구에서 사용하는 LDPC (low-density parity-check) 알고리즘이 비대칭적이고 가변적인 특성을 수용할 수 있다. 두개의 ECC 디코더와 한개의 인코드를 사용하는 이유는 다음과 같다. 스토리지에서 필요한 ECC 인코더의 성능은 스토리지 내부의 NAND 플래시 메모리의 칩의 개수에 비례하는 것으로 생각할 수 있는데 ECC 인코딩 시간은 일정하기 때문이다. 따라서 비교적 느리게 동작하는 NAND 플래시 메모리의 프로그램 시간을 고려하면 인코더의 개수는 1개로도 충분한 것으로 검토되었다. 그러나 ECC 디코더의 경우는 NAND 플래시 메모리의 읽기 시간이 비교적 짧기 때문에 더 많은 개수가

필요하다. 또한 LDPC 알고리즘이 확률적인 알고리즘이어서 수행시간이 가변적이므로, 디코더를 1개만 사용하는 경우는 병목 현상이 발생할 수 있다. 따라서 NAND 플래시 메모리의 읽기 시간과 병목 현상 방지를 고려하여 디코더는 2개를 사용한다.

4.2 펌웨어 (Firmware)

펌웨어는 주소 변환 기능을 통해 페이지 (또는 블록) 단위로 동작하는 NAND 플래시 메모리가 섹터 단위로 동작하는 스토리지로 동작할 수 있도록 한다. 펌웨어는 컨트롤러가 제공하는 백본 블록의 CPU와 SRAM을 이용하여 실행되며, 컨트롤러를 이용해 NAND 플래시 메모리를 비롯한 하드웨어를 제어한다. 펌웨어는 그림 18 에서 보이는 것과 같이 3 개의 계층 구조로 구성되며, 각 계층의 역할은 다음과 같다.

4.2.1 호스트 인터페이스 계층 (Host Interface Layer)

호스트 인터페이스 계층은 펌웨어 스택의 가장 상위에 있으며 3장에서 정의한 스토리지 인터페이스를 이용해서 호스트 시스템과 통신한다. 이 계층은 호스트 시스템으로부터 새로운 명령을 수신하고 하위 계층에 새로 수신한 명령의 실행을 요청한다. 그 후 하위 계층으로부터 요청했던 명령의 실행이 완료 통보를 받으면, 완료 결과를 호스트 시스템에 반환한다.

새로 수신된 명령이 존재하는지 확인하기 위해서는 스토리지 인터페이스의 제출 큐를 시작 (head) 부분 부터 확인한다. 만약 새로운 제출 큐 엔트리가 존재한다면, 해당 엔트리를 가져온다 (fetch). 엔트리에 포함된 명령을 해석한 후, 하위 계층인 플래시 변환 계층이 처리하기에 적합한

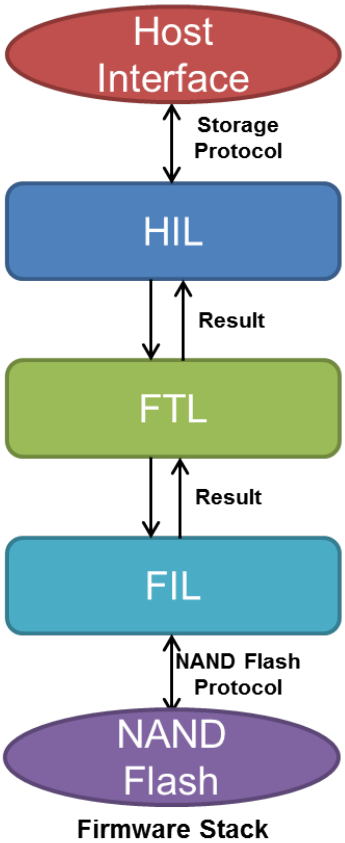

그림 18: 펌웨어 스택

내부 자료구조로 변환한다. 그리고 해당 자료구조를 플래시 변환 계층에 전달한다.

하위 계층, 즉 플래시 변환 계층이 명령 실행을 완료하면 호스트 인터페이스 계층에게 명령 실행 완료를 통보한다. 완료 통보를 받으면 호스트 인터페이스 계층은 스토리지 장치 인터페이스의 완료 큐에 추가할 엔트리를 준비한다. 해당 엔트리가 준비되면 완료 큐의 끝 (tail)에 엔트리를 추가하고, 인터럽트를 발생시켜서 명령 실행이 완료되었음을 호스트 시스템에게 통보한다.

본 연구의 호스트 인터페이스 계층은 호스트의 관리 명령은 자신이 직접 처리하고, 일반적인 I/O 명령 (Read, Write, Flush)은 하위 계층인 플래시 변환 계층이 처리하도록 전달한다. 호스트 인터페이스 계층이 제공하는 핵심적인 관리 명령으로는 스토리지에 대한 정보를 제공하는 Identify 명령이 있다. 호스트 시스템은 해당 명령을 통해 스토리지의 존재를 확인하고 용량 등의 특성을 확인할 수 있다.

4.2.2 플래시 변환 계층 (Flash Translation Layer)

플래시 변환 계층은 펌웨어 스택의 중간, 즉 상위 계층인 호스트 인터페이스 계층과 하위 계층인 플래시 인터페이스 계층의 사이에 위치한다. 이 계층은 호스트 인터페이스 계층에서 호스트 명령 처리 요청을 전달받아 플래시 변환 계층 내부에서 중간 처리 과정을 거친 후 플래시 인터페이스 계층으로 NAND 플래시 메모리에 대한 요청을 전달한다. 또한 자신에게 플래시 인터페이스 계층의 처리 결과가 반환되면, 중간 처리 과정을 거친 후 상위 계층인 호스트 인터페이스 계층으로 결과를 전달한다.

이 계층의 주 역할은 주소 변환 (Address Translation)이며 이를 위해 논리-물리 주소 매핑 정보를 유지한다. 호스트의 읽기 명령을 처리할 때는 매핑 정보를 참조하며, 쓰기 명령을 처리할 때는 매핑 정보를 갱신한다. 호스트 인터페이스 계층으로부터 읽기 명령을 수신하면, 플래시 변환 계층은 자신이 관리하는 논리-물리 주소 매핑 정보로부터 읽기 명령의 대상 논리 주소에 매핑된 NAND 플래시 메모리의 물리 주소를 확인한다. 이 확인 과정이 완료되면 NAND 플래시 메모리의 물리 주소를 포함한 읽기 명령을 하위 계층인 플래시 인터페이스로 전달한다.

호스트 인터페이스 계층으로부터 쓰기 명령을 수신하면, 해당 데이터가 프로그램 될 NAND 플래시 메모리의 물리 주소를 결정한 후 플래시

인터페이스 계층으로 프로그램 명령을 전달한다. 요청한 명령에 대해 플래시 인터페이스 계층이 처리 완료를 통보하면, 플래시 변환 계층도 호스트 인터페이스 계층에 처리 완료를 전달한다. 만약 호스트의 쓰기 명령에 대한 NAND 플래시 메모리의 프로그램이 완료되었다면, 플래시 변환 계층은 자신이 관리하는 논리-물리 주소 매핑정보를 새로운 NAND 플래시 메모리의 주소로 갱신한다.

그 외에도 가비지 컬렉션 (Garbage Collection), 마모 균등화 (Wear Leveling), 불량 블록 관리 (Bad Block Management) 등의 플래시 변환 계층에서 수행하는 일반적인 기능도 수행한다.

4.2.3 플래시 인터페이스 계층 (Flash Interface Layer)

플래시 인터페이스 계층은 펌웨어 스택의 최하단에 위치하면서 플래시 메모리 컨트롤러를 활용하여 NAND 플래시 메모리 연산을 수행한다. 이 계층은 상위 계층인 플래시 변환 계층으로부터 요청을 수신하면 해당 요청에 대응되는 NAND 플래시 메모리 명령으로 변환한 후 플래시 메모리 컨트롤러에 전송한다. 플래시 메모리 컨트롤러는 전송된 NAND 플래시 메모리 명령을 실행하고, 실행이 완료되면 플래시 인터페이스 계층에 명령 실행 완료를 통보한다. 플래시 메모리 인터페이스 계층이 NAND 플래시 메모리 명령 실행 완료를 통보받으면, 해당 명령의 실행 결과를 확인한다. 그 후 명령 실행 결과를 상위 계층인 플래시 변환 계층으로 전달한다.

이 과정에서 플래시 인터페이스 계층은 ECC 인코딩과 디코딩을 수행하게 된다. 쓰기 명령이라면 ECC 인코딩을 통해 패리티 정보를 계산해서 NAND 플래시 메모리의 스페어 영역에 계산된 패리티 정보를 추가한다. 읽기 명령의 경우는 NAND 플래시 메모리에서 읽어온 데이터를 ECC 디코딩을 통해 에러가 없는지 검증하고 필요한 경우는 패리티를 이용해

정정한다.

또한 플래시 인터페이스 계층은 배드 블록 발생을 탐지하여 상위 계층인 플래시 변환 계층에 통보해야 한다. NAND 플래시 메모리 오퍼레이션 중 erase, program 에서 에러가 발생하면 해당 블록은 배드 블록으로 간주해야 한다. 따라서 에러가 발생한 해당 블록에 대한 정보를 플래시 변환 계층에 통보하여 해당 블록이 더이상 사용되지 않도록 해야 한다. 특히 Program 오퍼레이션에서 에러가 발생하는 경우는 해당 데이터를 다른 페이지에 다시 프로그램 해야 하므로 해당 데이터가 program 오퍼레이션 완료시까지 보존될 수 있도록 대비가 필요하다.

제 5 장

호스트 시스템 설계

이 장에서는 제안된 스토리지 장치 인터페이스 및 스토리지 장치를 위한 호스트 시스템을 설계한다. 먼저 해당 스토리지 장치가 사용될 호스트 시스템 플랫폼 하드웨어를 설계한다. 그 후 해당 스토리지 장치를 활성화하는 소프트웨어인 부트 로더 (boot loader)와 디바이스 드라이버 (device driver)를 설계한다. 마지막으로 호스트 소프트웨어 수준의 최적화를 논의한다.

5.1 호스트 시스템 플랫폼 하드웨어

여기서는 제안된 스토리지 장치를 구동할 호스트 시스템 플랫폼 하드웨어를 설계한다. 3장에서 제안한 스토리지 장치 인터페이스는 기존 호스트 시스템의 DRAM 컨트롤러와 완벽하게 호환되므로 원칙적으로 호스트 시스템의 하드웨어를 변경 없이도 사용 가능하다. 그러나 기존의 플랫폼 하드웨어는 DRAM 컨트롤러와 제안된 스토리지 장치가 DRAM 버스를 통해 물리적으로 연결되어 있지 않으므로 이에 대한 연결이 필요하다. 또한 제안된 스토리지 장치가 명령 실행 완료 통보를 위한 인터럽트 신호를 전달할 수 있도록 호스트 시스템 CPU의 GPIO와의 연결이 필요하다.

본 연구에서는 신규 호스트 시스템 플랫폼 하드웨어를 처음부터 새로 설계하지 않고 기존 상용 태블릿 제품의 플랫폼으로 사용된 모바일 시스템 레퍼런스 보드 [40]의 설계를 인쇄 회로 기판 (PCB, Printed Circuit

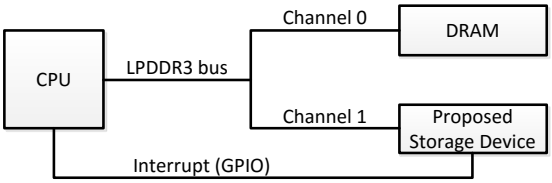

그림 19: 호스트 플랫폼 보드에서 DRAM 및 스토리지의 연결

Board) 수준에서 일부 수정하여 사용한다. 이를 통해 제안된 스토리지 장치 인터페이스 및 스토리지 장치의 설계와 구현에 집중하고 호스트 시스템의 구현에 필요한 시간과 노력을 최소화할 수 있다.

변경된 호스트 시스템 플랫폼 보드의 구성은 그림 19에 정리되어 있다. 호스트 시스템의 CPU는 ARM Cortex A15 듀얼 코어를 제공하는 Samsung Exynos 5250이며, 이 CPU는 2개의 LPDDR3 DRAM 채널을 제공한다. 제안된 스토리지 장치의 평가를 위해 그림 19에서와 같이 DRAM 채널 0은 실제 DRAM에 연결하고 DRAM 채널 1은 제안된 스토리지 장치에 연결하도록 플랫폼 보드를 설계하였다. 현재는 제안된 스토리지 장치로부터 부팅이 불가능하므로, SD 카드에 부트 로더 및 운영체제를 설치하여 평가를 진행했다.

5.2 부트 로더 (boot loader)

본 연구에서는 universal boot loader (U-Boot) [41]를 사용해서 시스템에 전원이 인가된 직후의 시스템 초기화 및 운영체제 (OS, Operating System)를 로딩한다. U-boot는 시스템 초기화 과정에서 DRAM 서브시스템도 초기화한다. 제안된 스토리지 장치 인터페이스는 호스트 시스템 관점에서 일반적인 DRAM과 동일하므로 부트 로더의 DRAM 서브시스

템 초기화 과정에서 해당 인터페이스도 일반적인 DRAM과 마찬가지로 초기화가 필요하다. 이를 위해 U-Boot를 변경하여 제안된 스토리지 장치 인터페이스에 대한 초기화도 수행되도록 한다. 여기에 필요한 변경은 크게 두 가지로 구성된다. 먼저 스토리지 장치 인터페이스의 공유 메모리 버퍼 영역을 시스템 메모리 맵에 반영해야 한다. 이 단계에서 기존 DRAM과의 인터리빙 정보를 설정해야 하는데, 이에 대해서는 6.3.2에서 논의한다. 그리고 공유 메모리 버퍼는 DRAM과 동일하게 간주되므로 DRAM과 마찬가지로 접근에 필요한 타이밍 정보를 설정해야 한다.

5.3 디바이스 드라이버

제안한 스토리지 장치를 운영체제에서 블록 장치로 사용하기 위해서는 그림 20에서 볼 수 있듯이 3장에서 제안한 프로토콜을 지원하는 새로운 디바이스 드라이버가 필요하다. 새로운 디바이스 드라이버는 Linux (커널 버전 3.4) 용으로 설계되었으며, 초기화, 스토리지 명령 제출, 스토 리지 명령 완료 처리 부분으로 구성된다.

본 장치의 디바이스 드라이버는 일반적인 Linux 블록 디바이스 드라이버와 마찬가지로 초기화 과정에서 스토리지 장치를 인식하고 장치 정보를 커널에 등록한다 [42] [43] [44]. 초기화의 첫 단계에서 스토리지 장치 인터페이스의 물리 주소를 가상 주소에 매핑한다. 제출 큐와 완료 큐 영역에 대해서는 `ioremap_nocache()` 커널 API를 호출하여 해당 가상 주소 영역을 캐싱 불가능한 영역으로 설정한다. 데이터 버퍼 영역에 대해서는 `ioremap()` 커널 API를 호출하여 해당 가상 주소 영역이 캐싱 가능하도록 설정하고, 필요한 경우에만 캐시 무효화 작업을 수행한다. 스토리지 장치 인터페이스에 대한 가상 주소 매핑이 완료되면 초기화 루

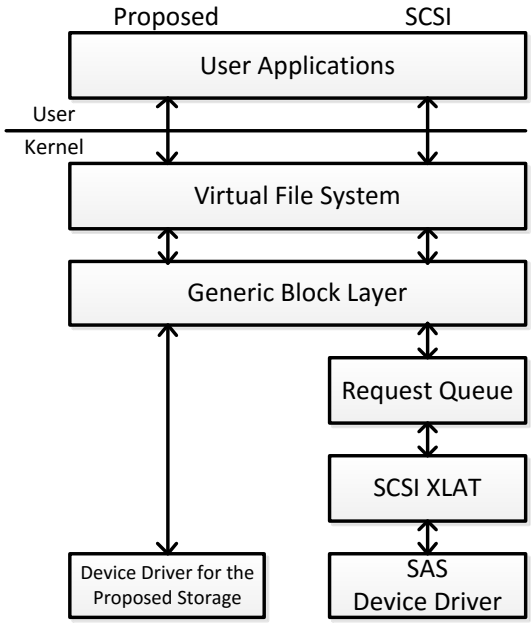

그림 20: Linux 스토리지 소프트웨어 스택

틴이 스토리지 장치에 관리 명령을 전송하여 스토리지 장치의 존재 여부, 용량, 기능 등을 식별한다.

스토리지 장치 식별이 완료되면, 초기화 루틴이 블록 장치 등록에 필요한 자료구조를 준비해서 `add_disk()` 커널 API를 호출해서 Linux 커널에 블록 장치로 등록한다. 이 시점부터 스토리지 장치에 SATA, SAS SSD와 같은 일반적인 블록 장치와 같은 방식으로 접근할 수 있다. 본 장치의 디바이스 드라이버는 상위 계층이 블록 I/O 요청을 전달하는 인터페이스로 `make_request()` API를 제공한다. 이 API는 블록 I/O 요청에 대한 입력으로 `bio` 구조체 (그림 21)를 전달받는다.

본 장치의 디바이스 드라이버는 *제출 처리기 (submission handler)*와

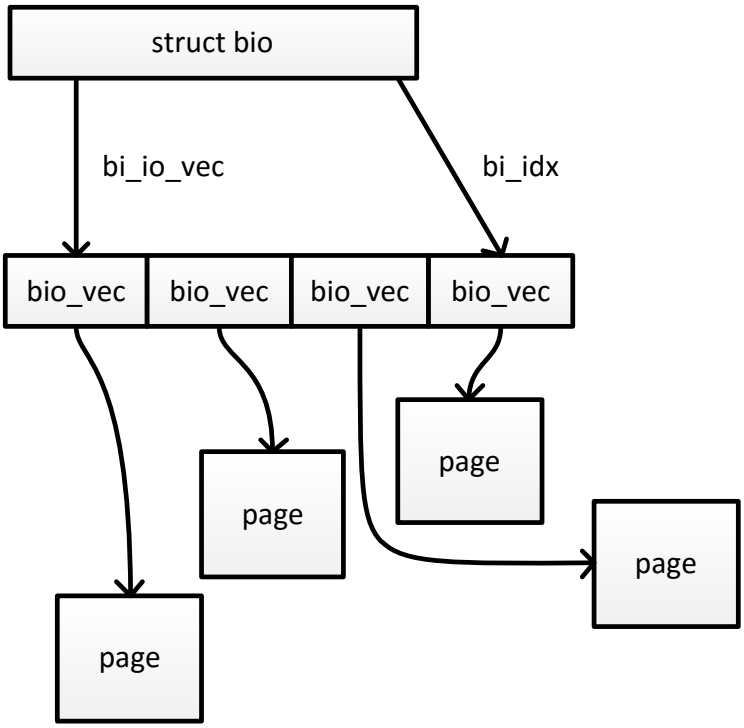

그림 21: bio 구조체

*완료 처리기 (completion handler)*를 이용해서 I/O 요청을 처리한다. 디바이스 드라이버에 상위 계층으로부터 (`make_request()` API를 통해서) 블록 I/O 요청이 전달되면, *제출 처리기*가 실행된다. *제출 처리기*는 `bio` 구조체 형태로 전달받은 블록 I/O 요청을 제안한 스토리지 장치 프로토콜에 맞도록 변환한다. 또한 해당 블록 I/O 요청을 위한 영역을 스토리지 장치 인터페이스의 데이터 버퍼 영역에 할당한다. 그리고 준비한 스토리지 명령이 포함된 제출 큐 엔트리를 제출 큐에 기록한다. 쓰기 명령의 경우는 제출 큐 엔트리를 기록하기 이전에 `bio` 구조체에 지정되어 있는 쓰기

데이터를 데이터 버퍼 영역에 복사해야 한다.

스토리지 명령의 완료는 *완료 처리기*에 의해 처리되며, 스토리지 장치가 생성한 완료 인터럽트를 수신하는 GPIO 핀의 인터럽트 처리기 (interrupt handler)에 의해 호출된다. *완료 처리기*는 완료 큐 엔트리를 확인해여 새롭게 완료된 스토리지 명령이 있을 경우 대응되는 `bio` 구조체와 연결된 Linux 커널의 상위 계층에 명령 실행 완료를 통보한다. 상위 계층으로의 통보는 `bio_end_io()` 커널 API를 통해서 가능하다. 읽기 명령이 완료된 경우는 스토리지 장치에서 읽어온 데이터가 저장되어 있는 데이터 버퍼 영역으로부터 `bio` 구조체에 지정된 버퍼 영역으로 데이터를 복사한다. 완료 처리기는 스토리지 장치가 인터럽트를 발생시킬 때 뿐 아니라 주기적으로 실행되도록 설정되어 있어서 인터럽트가 발생하지 않아도 완료된 명령을 처리할 수 있도록 설계하였다.

5.4 소프트웨어 최적화

디바이스 드라이버 설계시 적용 가능한 몇 가지의 소프트웨어 수준 최적화를 살펴본다.

5.4.1 DMA를 이용한 데이터 전송

데이터 버퍼와 `bio` 구조체의 버퍼 사이에서 데이터를 효과적으로 전송하기 위해 호스트 시스템의 CPU에 내장된 메모리 간 (memory-to-memory) DMA 장치를 활용할 수 있다 [39]. 메모리 간 DMA 장치가 제공되지 않는 호스트 시스템에서는 여전히 소프트웨어적인 메모리 복사 (Linux 커널의 `memcpy()` 또는 CPU에서 제공하는 인스트럭션)를 사용해야 한다. 소프트웨어적인 메모리 복사는 `load`와 `store` 연산을 계속해

서 데이터를 이동하므로 태생적으로 CPU에 많은 부하를 준다. 실험 결과 작은 크기의 무작위 I/O 성능은 DMA를 사용하는 경우가 소프트웨어적인 메모리 복사를 이용하는 경우에 비해 10배 이상 우수했다. 하지만 본 연구에서 사용한 호스트 시스템의 CPU에 내장된 DMA 장치는 스토리지 장치의 잠재적인 성능을 보여줄 수 있는 전송률을 제공하지 못했다. 이에 대해서는 6장에서 논의한다.

5.4.2 I/O 분할 처리

제안된 스토리지 장치의 디바이스 드라이버는 하나의 `bio` 요청을 작은 요청 크기를 가진 다수의 스토리지 명령으로 변환하여 처리한다. 이 방식에는 크게 두 가지 이유가 있다. 먼저, 현재 프로토콜은 분할 처리(scatter and gather)를 지원하지 않기 때문에 하나의 스토리지 명령을 처리하기 위해서는 스토리지 명령의 요청 크기에 해당되는 연속적인 공간을 데이터 버퍼 영역에서 확보해야 한다. 게다가 상위 계층이 데이터 버퍼 영역의 크기보다 더 큰 요청 크기를 갖는 `bio` 요청을 전달할 수도 있기 때문에 이 기법을 반드시 사용해야 한다.

두 번째 이유는 하나의 큰 I/O 요청을 작은 크기의 다수 I/O 요청을 분할하여 처리하는 것이 성능을 향상시킬 수도 있기 때문이다. 3.2에서 설명한 바와 같이, 스토리지 명령의 쓰기 또는 읽기 데이터는 각각 제출 큐 또는 완료 큐의 엔트리가 처리되기 전에 데이터 버퍼에 준비되어 있어야 한다.

그림 22의 상황을 예로 살펴보자. 디바이스 드라이버가 상위 계층으로부터 512 KB 데이터를 스토리지에 쓰라는 요청을 받았다. 여기서 두 가지 선택이 가능하다. 첫 번째 선택은 (A)와 같이 해당 요청을 하나의 스토리지 명령으로 처리하는 것이다. 이 경우 전체 512 KB 데이터를 데이터

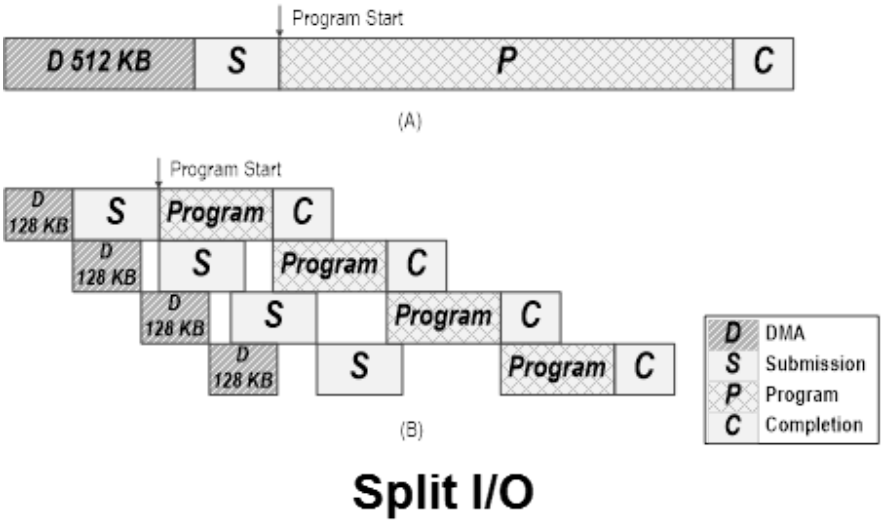

그림 22: I/O 분할 처리 예제

버퍼에 전송한 후 스토리지에 쓰기 명령을 전송할 수 있다. 두 번째 선택은 (B)와 같이 해당 요청을 4개의 128 KB 쓰기 요청으로 변환하여 처리하는 것이다. 이 경우 4개의 스토리지 명령은 각 명령에 대응되는 128 KB 데이터를 데이터 버퍼로 전송한 후에 스토리지로 전달할 수 있다. 이제 두 경우를 비교해보자. (A)의 경우는 스토리지 명령 1개로 처리되므로 명령 제출 및 완료 처리에 대한 비용이 적다. 하지만 실제로 스토리지 내부에서 NAND 플래시 메모리의 프로그램 연산이 시작되는 시점은 더 느리다. 그 이유는 전체 데이터 512 KB를 전송하는 시간이 오래 걸리기 때문이다. 따라서 스토리지 매체, NAND 플래시 메모리의 유휴 시간이 더 길어진다. 스토리지 매체의 대역폭이 호스트 인터페이스 대역폭보다 작은 이와 같은 경우는 스토리지 매체의 유휴시간을 줄이고 사용률을 가능한 높이는 것이 성능 향상에 유리하다.

5.4.3 캐시 무효화 비용 최소화

제안한 디바이스 드라이버 설계에서는 데이터 버퍼 영역은 기본적으로 캐싱 가능한 영역으로 설정하고, DMA 전송 이전과 이후에 각각 캐시 무효화를 수행한다. 본 연구의 호스트 시스템 플랫폼에서 512 KB 영역에 대한 캐시 무효화 시간은 약 290 *us*로 측정되었고, 이는 상당한 긴 시간이다. 따라서 캐시 무효화 시간을 감소시킬 필요가 있다. 이를 위해서 하나의 큰 영역에 대한 DMA를 수행하기 보다는, 하나의 큰 DMA를 다수의 작은 크기의 DMA로 나누고, 캐시 무효화 동작을 파이프라인 방식으로 처리하도록 한다.

5.4.4 데이터 버퍼의 외부 단편화 최소화

스토리지 명령이 계속 수행되면서 데이터 버퍼의 할당 및 해제로 인해 데이터 버퍼 영역이 단편화된다. 데이터 버퍼 영역에 대한 외부 단편화는 다음 두 가지 이유 때문에 최소화되어야 한다. 먼저, 현재 프로토콜 설계는 한 개의 제출 큐 엔트리에 한 개의 버퍼 인덱스만 지정할 수 있다. 따라서 스토리지 명령에 대한 데이터 버퍼는 연속적으로 할당되어야 한다. 두 번째 이유는 데이터 버퍼의 단편화로 인해 데이터 버퍼 영역을 연속적으로 할당할 수 없는 경우, `bio` 요청을 다수의 매우 작은 크기를 갖는 스토리지 명령으로 분할 처리해야 하기 때문이다. 5.4.2에서 단일 I/O를 다수의 작은 크기를 갖는 I/O 로 분할하여 처리하는 것이 성능을 향상시킬 수 있다는 것을 확인했다. 그러나 다수의 스토리지 명령을 처리하는 비용이 이 스토리지 장치의 병렬성 활용으로 인한 이득보다 커지는 경우 전체 성능이 하락한다.

현재 프로토콜 설계상의 문제로 인해 발생하는 버퍼 단편화를 최소

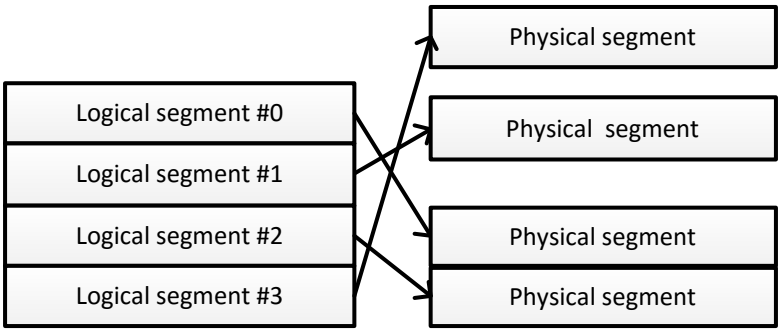

그림 23: Scatter-gather IO

화하기 위해 잘 알려진 버디 메모리 할당 알고리즘 [45] [42]을 사용할 수 있다. 또한 이 문제는 프로토콜을 수정하여 제출 큐 엔트리에 다수의 버퍼 인덱스를 지정할 수 있도록 하는 분할 처리 (scatter and gather) (그림 23 참고) 를 지원하면 해결될 것으로 예상된다.

5.4.5 Linux I/O 스케쥴러 미사용

제안된 장치의 디바이스 드라이버는 그림 20의 범용 블록 계층 (generic block layer)에서 제공하는 I/O 스케쥴러를 사용하지 않는다. 이것은 상위 계층의 I/O 요청이 I/O 스케쥴러를 거치지 않고 제안된 장치로 빨리 전달되어 스토리지 관점에서 최적화된 스케쥴링을 통해 빨리 처리되도록 하기 위한 것이다. 유사한 이유로 NVMe 디바이스 드라이버도 I/O 스케쥴러를 사용하지 않는다.

I/O 스케쥴러는 상위 계층의 I/O 요청을 자신의 큐에 추가시켜서 병합 (merge), 재정렬 (reordering)등의 작업을 거쳐 스토리지에 전달한다. 주 목적은 병합을 통해 스토리지로 전달되는 I/O 요청의 개수를 줄이고,

재정렬을 통해 탐색 거리 (seek distance)를 줄여서 스토리지 I/O 성능을 향상시키는 것이다. 그러나 이것은 탐색 시간이 수 *ms* 수준인 하드디스크를 사용하는 경우엔 유효하나 탐색 시간이 없는 NAND 플래시 메모리 기반 스토리지의 경우에는 큰 의미가 없다.

오히려 I/O 스케쥴러의 큐에서 대기하는 시간으로 인해 스토리지로의 I/O 요청 전달이 늦어져서 전체적인 응답시간이 증가하는 역효과를 가져오기도 한다. 따라서 가능한 빠르게 스토리지에 I/O 요청을 전달하고, 스토리지가 자신의 관점에서 최적의 스케쥴링을 통해 I/O 요청을 처리하는 것이 전체 처리량을 개선하는데 도움이 된다.

단, 전체 처리량 개선이 개별 스토리지 I/O 관점에서는 단점이 될 수도 있다. 예를 들면 사용자가 문서를 읽기 위해 스토리지에 읽기 I/O를 요청했으나, 스토리지 자체의 스케쥴링 정책에 의해 해당 읽기 I/O에 대한 우선순위가 낮아 늦게 처리되어 사용자 경험이 떨어지는 상황이 발생할 수 있다. 이와 같은 경우 I/O 스케쥴러는 사용하는 것이 장점이 될 수도 있다. 왜냐하면 I/O 스케쥴러는 호스트 시스템에서 동작하므로 호스트 시스템의 컨텍스트를 알고 있어서 어떤 I/O 요청이 긴급하게 처리되어야 하는지 알 수 있고 이러한 I/O 요청을 먼저 수행할수 있기 때문이다. 이러한 문제를 해결하기 위해 호스트 시스템이 I/O 요청시에 이와 같은 우선순위에 대한 힌트를 스토리지에 전달한다면 복잡한 I/O 스케쥴러를 사용하지 않고도 호스트가 원하는 우선순위대로 스토리지가 I/O 요청을 수행하면서도 전체 처리량을 개선할 수 있을 것으로 기대된다.

제 6 장

평가

이 장에서는 3 장, 4 장, 5장의 설계를 바탕으로 구현된 스토리지를 평가한다. 먼저 호스트 시스템의 사양과 실험 받법 등을 포함한 평가 환경을 기술한다. 평가 환경을 기반으로 새로 제안된 스토리지 프로토콜의 오버헤드, 제안된 스토리지의 성능 등에 대해 정량적인 분석을 실시한다. 마지막으로 정성적 평가를 실시하여 스토리지를 더욱 효과적으로 활용하기 위한 방안을 검토한다.

6.1 평가 환경

앞에서 제안한 스토리지와 호스트 시스템을 구현한 후 통합하여 실제 사용 가능한 모바일 컴퓨터 시스템을 구축했다. 호스트 시스템의 운영체제는 안드로이드를 사용하였다. 구현된 전체 시스템에 대한 정보는 표 3에 정리되어 있고, 스토리지 컨트롤러의 정보는 표 4에 정리되어 있다.

현재 호스트 시스템에서는 본 논문에서 제안한 스토리지에서 부팅이

어플리케이션 프로세서	Samsung Exynos 5250 (ARM Cortex-A15 dual @ 1.7 GHz)
DRAM 인터페이스	LPDDR3, 2 채널 (600 MHz)
DRAM 용량	1.5 GB (DRAM 채널 0)
스토리지 용량	32 GB (DRAM 채널 1)

표 3: 프로토타입 시스템 요약

Fabrication 공정	28 nm CMOS
Controller 패키지 크기	20.9 mm^2 (5.5 mm x 3.8 mm)
eMCP 패키지 크기	11.5 mm x 13.0 mm x 1.2 mm
Core clock	266 MHz
Host interface clock	800 MHz (1,600 Mb/s)
Host interface clock	800 MHz (1,600 Mb/s)
NAND flash interface clock	233 MHz (466 Mb/s)
전력 소비량	순차 읽기 560 mW, 순차 쓰기 450 mW

표 4: 컨트롤러 칩 사양

불가능하기 때문에 별도의 스토리지 장치 (SD 카드)를 이용해서 부팅한다. 이 방식의 장점은 운영체제와 관련된 예상 불가능한 I/O 요청이 SD 카드에서 처리되기 때문에 평가중인 본 논문의 스토리지 장치에 영향을 주지 않는다는 것이다. 구현된 스토리지 장치는 전체 용량을 단일 ext4 파일시스템 파티션으로 설정했다.

이 장에서 제시하는 평가 결과는 모두 실제 시스템에서 측정한 것이다. 성능 측정은 디바이스 드라이버에서 트레이스를 수집하거나 iozone 벤치마크를 이용했다. 또한 평가를 위해 마이크로 벤치마크를 사용하는데, 이는 스토리지 장치 자체의 성능 특성 분석에 집중하기 위해서이다.

이 장의 평가 내용에는 포함하지 않았으나 제안된 스토리지에 운영체제 (안드로이드)를 설치하고 운영하는 경우도 잘 동작했다. 단, 이 경우에도 운영체제 부팅에 필요한 커널 이미지와 루트 디스크는 SD카드에 저장했다.

6.2 정량적 평가

이 섹션에서는 3 장, 4 장, 5 장의 설계를 바탕으로 구현된 스토리지를 정량적으로 평가한다. 먼저 호스트 시스템의 사양과 실험 받법 등을 포함한 평가 환경을 검토한다. 그 후 새로 제안한 프로토콜의 오버헤드를 평가함으로써 타당성을 검증한다. 또한 제안된 스토리지 인터페이스는 제한된 인터페이스 버퍼 크기를 가지므로 이에 대한 민감도 분석을 실시한다. 그리고 스토리지의 읽기/쓰기 성능을 평가한다. 마지막으로 제안된 스토리지의 성능을 최신 UFS 2.0 장치와 비교함으로써 제안된 스토리지의 현재 수준을 확인한다.

6.2.1 프로토콜 오버헤드

여기서는 본 연구에서 제안한 프로토콜의 타당성을 검증하기 위해 사용자 데이터를 전송하기 위해 지불해야 하는 오버헤드를 평가한다. 먼저 오버헤드를 수치화 하기 위해 스토리지 프로토콜 오버헤드를 다음과 같이 정의한다.

$$storage\ protocol\ overhead = 1 - \frac{protocol\ bandwidth}{interface\ bandwidth}$$

여기서 인터페이스 대역폭 ($interface\ bandwidth$) 데이터 전송 방식 (DMA 또는 소프트웨어적인 메모리 복사)에 의해 결정되며, 호스트 시스템에서 사용할 수 있는 데이터 전송 방식에 따라 달라진다. 인터페이스 대역폭으로 DRAM 인터페이스, 구체적으로 LPDDR3 인터페이스의 대역폭을 고려하지 않는 이유는 실제 데이터 전송률은 데이터 전송 방식에 따라 달라지고 이것이 해당 시스템에서의 실질적인 최대 대역폭이기 때문이다. 실제로 본 연구에서 사용한 시스템은 LPDDR3 인터페이스의 대역폭에 비해 DMA 의 대역폭이 매우 낮아서 스토리지의 잠재 성능을 평가할

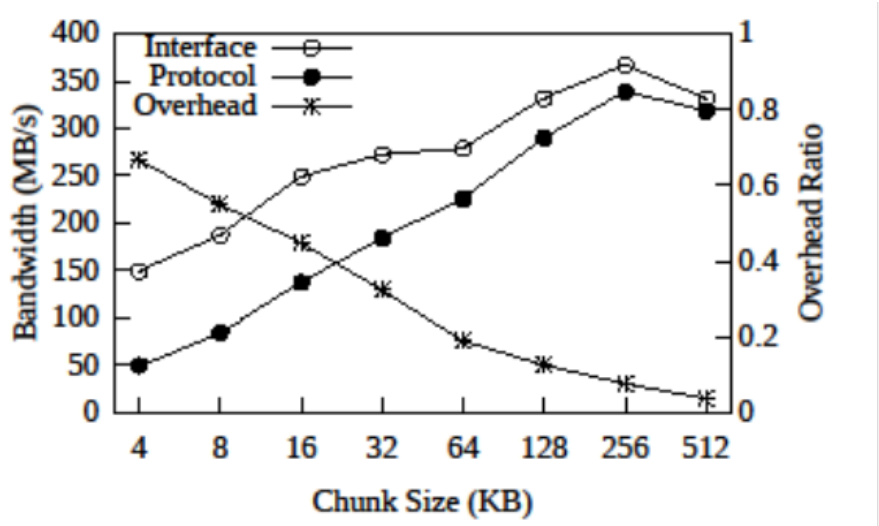

그림 24: 인터페이스 대역폭과 프로토콜 대역폭

수 없었다. 프로토콜 대역폭 (*protocol bandwidth*)은 스토리지 명령의 제출 및 완료에 대한 처리를 포함한 대역폭이다. 당연히 프로토콜 대역폭은 인터페이스 대역폭보다 낮을 수 밖에 없다.

프로토콜 오버헤드는 유저 스페이스에서 QD1으로 스토리지에 읽기 요청을 전송한 후, 디바이스 드라이버에서 이벤트 트레이스를 수집하고 이것을 기반으로 인터페이스 대역폭과 프로토콜 대역폭을 분석하여 계산했다. 또한 측정의 정확도를 높이기 위해 스토리지 펌웨어가 호스트의 읽기 명령을 처리할 때 실제로 NAND 플래시 메모리에 접근하지 않고 바로 결과를 반환하도록 수정했다. 따라서 NAND 플래시 메모리에 접근하는 시간이 제외되므로 보다 정확한 측정이 가능하다.

프로토콜 오버헤드 평가 결과는 그림 24에 제시되어 있다. 이 그래프는 각 요청 크기에 대한 인터페이스 대역폭과 프로토콜 대역폭을 보여준다. 현재 환경에서는 512 KB 에 대한 읽기 명령의 인터페이스 대역폭은

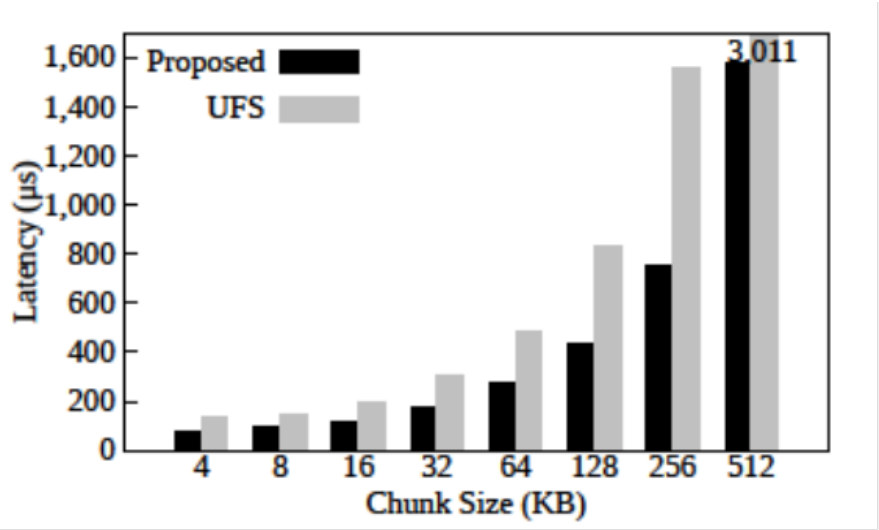

그림 25: 프로토콜 처리를 위한 오버헤드 레이턴시

366.7 MB/s 이다. 본 평가에서는 분석의 복잡도를 낮추기 위해 QD1 조건에서 실험을 진행해서 인터페이스 대역폭이 QD4 조건에 비해 다소 낮게 측정되었다. 또한 요청 크기가 증가할수록 상대적인 오버헤드의 비중이 감소하는 것을 볼 수 있다. 최소, 최대 오버헤드 비율은 각각 3.8 % (512 KB 요청 크기)와 66.8 % (4 KB 요청 크기)이다.

그림 25에 제안된 스토리지와 상용 UFS 2.0 장치에서 측정한 프로토콜 오버헤드에 해당되는 레이턴시가 비교되어 있다. 모든 요청 크기에서 제안된 스토리지 프로토콜의 오버헤드가 UFS 프로토콜의 오버헤드보다 낮지만, 완전히 동일한 조건이 아니므로 1:1 비교는 의미가 없다. 앞에서 기술한 것과 같이 읽기 요청의 처리 시간에서 NAND 플래시 메모리 접근 시간을 제외하기 위해서 제안된 스토리지의 펌웨어를 수정하여 읽기 요청을 처리할 때 NAND 플래시 메모리에 접근하지 않고 호스트 인터페이스 계층에서 바로 (무의미한) 읽기 데이터를 반환하도록 했다. 그러나

UFS 장치의 펌웨어를 같은 조건으로 수정할 수는 없기 때문에 UFS 장치의 테스트 대상 영역에 DISCARD 명령을 수행하여 매핑 정보를 제거한 후 측정을 진행했다. UFS 장치의 펌웨어 내부 구조를 알수는 없으나, 제안된 펌웨어와 비슷한 계층 구조로 되어 있다고 가정하면 읽기 명령을 처리하기 위해 호스트 인터페이스 계층 및 플래시 변환 계층까지 실행된 후 결과를 반환했을 것으로 추정된다.

6.2.2 스토리지 인터페이스 버퍼 크기와 최대 전송 속도

여기서는 스토리지 인터페이스의 버퍼 크기와 최대 전송 속도간의 관계를 살펴본다. 이 관계를 살펴보는 이유는 인터페이스 버퍼 크기가 스토리지 인터페이스의 성능을 결정하는 주 요소이기 때문이다.

평가를 위해 인터페이스 버퍼 크기 (X 축)와 요청 크기 (서로 다른 선)에 대한 스토리지 장치의 QD4에서의 읽기 성능을 측정했다. 이 평가에서도 앞의 프로토콜 오버헤드 평가와 마찬가지로 펌웨어를 수정하여 호스트의 읽기 요청에 대해 NAND 플래시 메모리에 대한 접근 없이 바로 (의미없는) 읽기 데이터를 반환하도록 했다. 따라서 이 결과는 NAND 플래시 메모리의 성능을 제외하고 순수하게 스토리지 인터페이스 버퍼 크기와 최대 전송 속도간의 관계에 대한 것이다.

평가 결과는 그림 26에 정리되어 있다. 평가 결과를 보면 256 KB 요청 크기에서 436 MB/s 의 최대 성능을 얻었다. 측정한 모든 요청 크기에서 512 KB 를 초과하는 인터페이스 버퍼 크기는 추가 성능 향상을 얻지 못했다. 또한 작은 요청 크기에서는 최대 성능도 작은 버퍼 크기에서 달성됨을 알 수 있다.

흥미로운 점은 버퍼 크기가 128 KB 일 때, 256 KB 요청 크기의 성능이 64 KB 요청 크기의 성능보다 낮다는 것이다. 이러한 비직관적인 결과

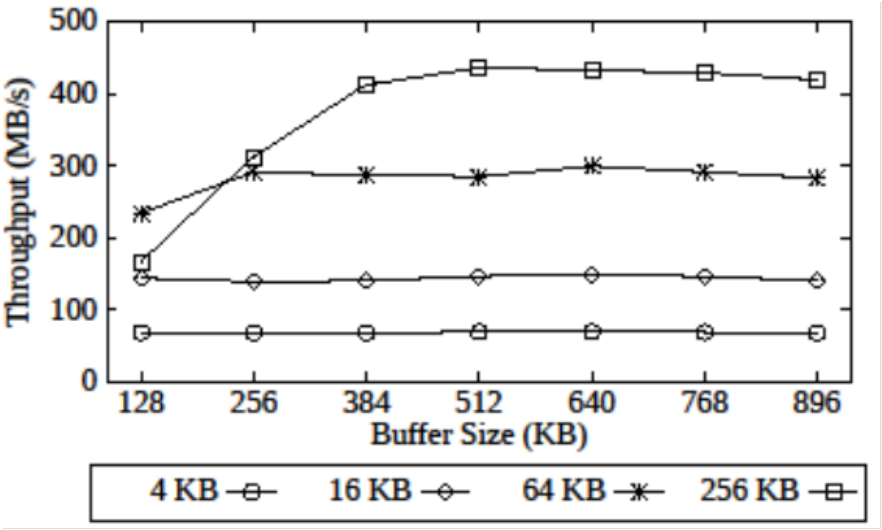

그림 26: 버퍼 용량과 요청 크기의 변동에 따른 최대 대역폭 (QD4)

의 원인은 ① 호스트의 대기 큐에서의 대기 시간 증가, ② 버퍼의 독점으로 인한 병렬성 저하로 추정된다. 이 현상은 다음의 그림 27의 시나리오를 통해 설명할 수 있다. 문제를 간단하게 하기 위해 2개의 쓰레드만 존재하는 것으로 가정하자. 먼저 64 KB 요청 크기에 대해 살펴보자. 현재 데이터 버퍼는 128 KB 모두 비어 있으므로, 쓰레드 0 (T0)은 데이터 버퍼 중 64 KB를 할당받아서 읽기 작업을 제출할 수 있다. 쓰레드 1 (T1) 역시 나머지 데이터 버퍼 64 KB를 할당받아서 읽기 작업을 제출할 수 있다. T0의 첫 64 KB에 대한 읽기 작업이 종료되면 T0는 나머지 64 KB에 대한 읽기 작업을 제출하려고 시도한다. 현재 데이터 버퍼의 64 KB가 유휴 상태이므로 T0는 기다리지 않고 읽기 작업을 제출할 수 있다. 한편 T1도 첫번째 읽기 작업을 완료하고 데이터 버퍼를 할당받기 위한 대기시간 없이 나머지 64 KB에 대한 읽기 작업을 바로 제출할 수 있다.

이제는 요청 크기가 128 KB 인 경우를 살펴보자. T0는 대기시간 없

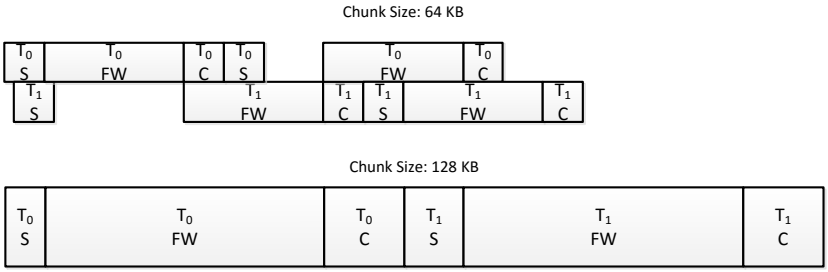

그림 27: 버퍼 용량 부족으로 인한 성능 저하

이 바로 128 KB 읽기 요청을 제출할 수 있다. 그러나 T1은 데이터 버퍼를 할당받기 위해 T0의 읽기가 완료될 때 까지 기다려야 한다. 따라서 전체 수행 시간은 그림 27에서 보는 것과 같이 요청 크기가 64 KB 인 경우가 더 짧다.

현재의 프로토콜에서 발생하고 있는 이 문제는 버퍼에서 데이터 전송이 발생하지 않는 상황에서도 각 스토리지 명령이 불필요하게 버퍼를 점유하고 있기 때문이다. 이 문제를 개선하기 위해서는 각 명령의 버퍼 점유 시간을 최소화해야 한다. 가령 읽기 명령에 대해서는 명령 제출 시점이 아닌 NAND 플래시 메모리에서 데이터를 가져오는 시점에서 데이터 버퍼를 할당하고, 쓰기 명령에 대해서는 NAND 플래시 메모리로 데이터를 전송한 이후에는 즉시 데이터 버퍼를 해제하는 방법을 사용할 수 있을 것이다.

데이터 버퍼 사용 시간을 최소화하는 기법을 적용하면 스토리지 인터페이스가 제공하고자 하는 목표 성능을 달성하기 위한 데이터 버퍼 필요량을 다음과 같이 계산할 수 있다. 4 KB 데이터 버퍼로 데이터를 넣고 꺼내는 시간을 각각 10 us 라고 가정하면, 1초에는 50000번의 I/O를 수행할 수 있다. 이를 대역폭으로 환산하면 200 MB/s 가 되고, 만약 128 KB의

버퍼가 있다면 25600 MB/s, 즉 25.6 GB/s 의 대역폭을 확보할 수 있다. 버퍼에 데이터를 넣는 작업과 꺼내는 작업을 중첩시키면 같은 양의 데이터 버퍼로도 더욱 빠른 대역폭을 확보할 수 있다.

쓰기 명령의 성능은 인터페이스 버퍼 용량보다는 NAND 플래시 메모리 자체의 쓰기 대역폭에 크게 연관되므로 이에 대한 평가는 생략한다.

6.2.3 읽기/쓰기 성능

제안된 스토리지의 프로토콜의 오버헤드에 대한 평가와 스토리지 인터페이스 버퍼 크기에 대한 민감도 분석을 완료햇으므로, 이제는 제안된 스토리지 자체의 성능을 평가한다. 스토리지 장치의 가장 기본적인 성능은 읽기/쓰기 성능이며, 평가는 널리 알려진 벤치마크 프로그램인 *IOZone file system benchmark* [46]를 이용했다.

`$ iozone -ecI -+n -L64 -S32 -r(CHUNKSIZE)k -s1024m -i0 -i1 -t1` // QD 1

`$ iozone -ecI -+n -L64 -S32 -r(CHUNKSIZE)k -s1024m -i0 -i1 -t4` // QD 4

평가를 위해 스토리지의 전체 용량을 단일 ext4 파일시스템으로 만들고 다음의 파라미터를 사용해서 QD1, QD4 성능을 측정했다.

성능 측정 결과는 그림 28에 제시되어 있다. 읽기의 경우 QD1, QD4 모두 요청 크기가 증가함에 따라 전송 속도도 계속해서 증가하여, 요청 크기 512 KB와 QD4 조건에서 최대 전송 속도가 432.2 MB/s 로 측정되었다. 쓰기의 경우도 읽기와 마찬가지로 QD1, QD4 모두 요청 크기가 증가함에 따라 전송 속도도 계속해서 증가한다. 그러나 QD4의 경우 요청 크기 64 KB에서 성능이 거의 포화됨을 알 수 있다. QD1의 경우는 계속해서 성능이 증가하다가 요청 크기 256 KB 에서 성능이 거의 포화된다. 최대 쓰기

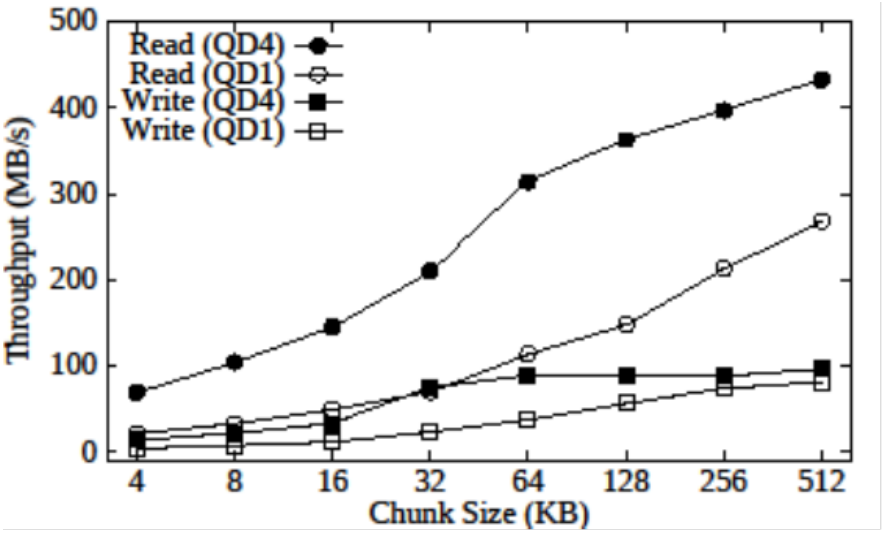

그림 28: 다양한 요청 크기에서의 읽기/쓰기 성능

속도는 요청 크기 512 KB와 QD4 조건에서 95.8 MB/s 이다.

6.2.4 성능 비교

이제는 제안된 스토리지의 성능을 최신 스마트폰에 탑재된 UFS의 성능과 비교한다. 그러나 이 비교의 목적은 두 플랫폼 또는 스토리지 장치를 1:1로 비교하기 위한 것은 아니며, 단지 비교를 통해 제안된 스토리지의

	본 연구	UFS (Galaxy S6)
어플리케이션 프로세서	Exynos 5250 (ARM Cortex A15 * 2 @ 1.7 GHz)	Exynos 7420 (ARM Cortex A57 * 4 + A53 * 4 @ 2.1 GHz)
메인 메모리	1.5 GB (LPDDR3 @ 600 MHz)	3 GB (LPDDR4 @ 1,555 MHz)
스토리지	32 GB	64 GB

표 5: 본 연구와 UFS 플랫폼의 사양 비교

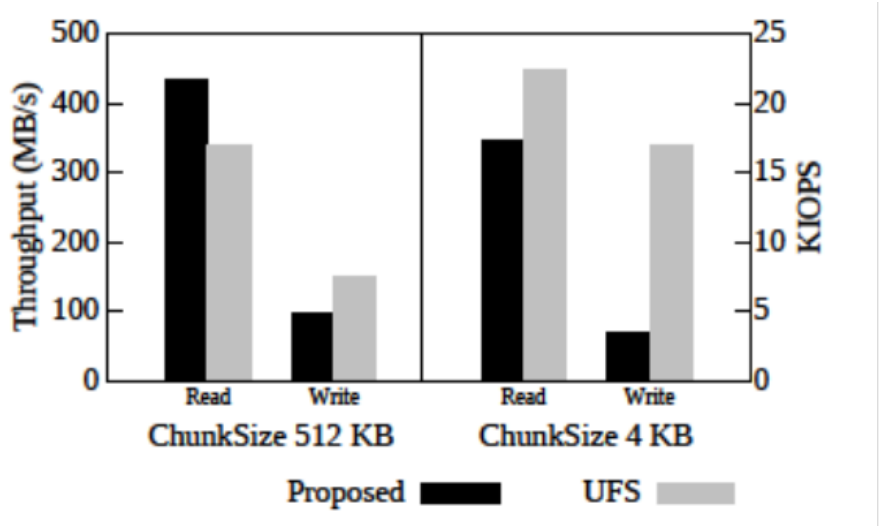

그림 29: UFS 스토리지와의 성능 비교

장단점 및 현재 수준을 파악하기 위한 것이다.

표 5에 스토리지의 사양을 비롯한 전체 플랫폼의 사양이 정리되어 있는데, UFS의 호스트 시스템이 제안된 스토리지의 호스트 시스템보다 훨씬 고사양임을 알 수 있다. 주어진 사양을 보면 UFS의 호스트 시스템이 CPU, 메인 메모리의 용량과 속도, 스토리지의 용량을 비롯한 모든 면에서 우월하다. 또한 보이지 않는 부분에서 우월한 점도 있을 것으로 추정되는데, 그것은 UFS 장치가 상용화된 제품이므로 상당한 수준의 최적화가 적용되었을 것이기 때문이다. 반면 본 연구에서 제안된 스토리지는 아직 프로토타입 수준이므로 많은 최적화가 필요하다. 추정 가능한 또다른 사항은 UFS가 제안된 스토리지 장치에 비해 용량이 2배이므로 높은 병렬성으로 인해 스토리지 매체 수준에서의 대역폭이 2배일 것이라는 점이다. 따라서 제안된 스토리지 성능이 UFS에 비해 떨어지더라도 당연한 결과로 간주할 수 있다.

그림 29의 결과는 iozone을 통해 측정한 QD4 의 성능을 비교한 것이다. 제안된 스토리지 장치는 많은 불리한 점, 특히 용량이 1/2에 불과함에도 불구하고, 순차 읽기에서 UFS 장치보다 100 MB/s 정도 더 우월한 성능을 보인다. 이것은 앞에서 평가된 바와 같이 프로토콜 오버헤드가 낮기 때문인 것으로 추정된다. 512 KB 쓰기와 4 KB 읽기의 경우는 제안된 스토리지의 성능이 낮은데, 이것은 스토리지 장치의 용량이 UFS 에 비해 절반이기 때문에 예측 가능한 결과이다. 원인을 분석해보면 쓰기 작업은 일반적으로 NAND 플래시 메모리 성능과 칩의 개수에 비례하고, 랜덤 읽기 성능 역시 칩이 개수에 비례하기 때문인 것으로 추정된다. 마지막으로 4 KB 쓰기 성능의 열세는 병렬성의 열세와 캐시 쓰기 기능의 미지원 때문이다. 캐시 쓰기 기능이 지원되면 실제 NAND 플래시 메모리에 프로그램이 완료되지 않아도 호스트 시스템에 쓰기 명령이 완료되었다고 통보한다. 이렇게 함으로써 호스트 시스템은 NAND 플래시 메모리의 긴 프로그램 시간을 기다리지 않고 다음 스토리지 작업을 진행할 수 있다. 단, 중간에 예상치 못하게 전원 공급이 차단되면 실제로 프로그램이 완료되지 않은 데이터는 손실될 가능성이 있으므로 주의해야 한다.

결과를 요약하면 현재는 최신 UFS 장치에 비해 순차 읽기 성능만 우월하고 나머지 3개 성능은 UFS 장치보다 떨어진다. 그러나 동일한 호스트 시스템 환경과 NAND 플래시 메모리 용량을 갖추고 캐시 쓰기 기능을 구현한다면 더 우월한 성능을 보일 수 있을 것으로 예상된다. 또한 상용 제품 수준의 최적화도 동반되어야 할 것이다.

6.3 정성적 평가

이 장에서는 제안된 스토리지 아키텍쳐를 구현하면서 발견한 문제점과, 해당 아키텍쳐를 효과적으로 활용하기 위한 제안을 논의한다.

6.3.1 DRAM 버스 대역폭 사용량 증가

본 연구에서 제안한 스토리지 인터페이스는 DRAM 인터페이스를 기반으로 하므로 전용 버스가 있는 기존의 스토리지 인터페이스에 비해 DRAM 버스 대역폭을 더 많이 사용한다. 읽기 명령을 처리하는 상황을 가정해보자. 스토리지가 NAND 플래시 메모리로부터 요청된 데이터를 읽어서 데이터 버퍼에 준비해두었다. 호스트 시스템은 DRAM 인터페이스상에서 DMA 를 수행하여 준비된 데이터를 가져갈 것이다. 여기서 DRAM 버스 대역폭을 한 번 사용한다. DMA 장치는 데이터 버퍼에서 읽어온 데이터를 최종 목적지인 `bio` 버퍼에 저장할 것이다. 여기서 DRAM 버스 대역폭을 두 번째로 사용한다. 기존의 eMMC나 UFS와 같이 스토리지 전용 버스를 사용하는 인터페이스에서는 DRAM 버스 대역폭을 한 번만 사용한다. 이처럼 두 배로 늘어난 DRAM 버스 대역폭 사용량은 스토리지가 사용하는 대역폭이 적은 경우는 문제가 되지 않으나, 많은 대역폭을 사용하는 경우에는 문제가 발생할 가능성이 있으므로 추가 연구가 필요하다.

6.3.2 유연한 DRAM 주소 매핑

현대의 컴퓨터 시스템은 보통 높은 대역폭을 제공하는 메모리 서브시스템을 구현하기 위해 주소 인터리빙 기법을 사용한다 [9]. 인터리빙 기법이 적용되면, 메인 메모리의 물리 주소 공간이 다수의 DRAM 채널, 다수의 DRAM 랭크/뱅크에 대해 실제 메모리 용량보다 작은 단위로 분할되어

매핑된다. 예를 들어 n개의 뱅크로 구성된 메모리가 있다고 가정하면, 메모리 주소 공간을 n개의 뱅크에 순서대로 인터리빙되도록 설정함으로써 병렬성을 높여 성능을 향상시킬 수 있다.

그런데 제안된 스토리지 장치를 사용하기 위해서는 DRAM 인터리빙과 관련하여 검토할 점이 있다. 제안된 스토리지 장치 인터페이스가 DRAM 주소 공간에 매핑되지만 이 영역은 인터리빙에 참여해서는 안된다. 그 이유는 이 영역이 일반적인 DRAM과 달리 스토리지 장치 인터페이스 전용 영역이기 때문이다. 따라서 해당 영역을 DRAM 인터리빙에서 제외시켜야 한다. 그런데 일반적인 컴퓨터 시스템은 전체 채널, 랭크, 뱅크가 DRAM 인터리빙에 참여하도록 설계되어 있어서, 본 스토리지 장치를 사용하는 경우와 같이 특정 채널, 특정 랭크, 특정 뱅크만 DRAM 인터리빙에 불참하도록 설정하기가 어려운 문제가 있다.

위의 문제를 해결하기 위해서는 호스트 시스템에서 보다 유연한 DRAM 인터리빙과 매핑 방식을 지원해야 한다. 예를 들어 다음의 두 가지 상황을 고려해보자: ① 제안된 스토리지 인터페이스가 인터리빙에 참여하는 경우 ② 제안된 스토리지 인터페이스가 인터리빙에 참여하지 않는 경우. 먼저 첫 번째 경우는 다음과 같은 이유로 인터리빙 단위가 가변적인 것이 좋다. 인터리빙 단위 큰 단위로 고정되어 있는 경우를 살펴보자. 이 경우는 스토리지 인터페이스가 자신에게 할당된 주소 공간의 극히 일부만을 사용하게 된다. 스토리지 인터페이스 관점에서는 호스트 시스템에게 제공하는 인터페이스 주소 공간 전체가 호스트 시스템의 주소 공간에 매핑되므로 문제가 없다. 그러나 호스트 시스템 관점에서는 스토리지 인터페이스에 할당되었으나 실제로 사용되지 않는 영역에 대한 주소 공간이 낭비된다.

이번에는 인터리빙 단위가 아주 작은 단위로 (캐시 블록 크기 정도) 고정된 경우를 가정해보자. 이 경우는 스토리지 인터페이스가 호스트 시

스템의 주소 공간에 아주 작은 크기로 분할되어 매핑된다. 결국 호스트 시스템의 디바이스 드라이버가 인터리빙 단위에 맞춰서 스토리지 인터페이스의 데이터 버퍼에 대한 데이터 전송 작업을 수행해야 하고 이는 연속적인 영역에 대해 데이터 전송을 수행하는 것 보다 성능이 현저히 떨어질 가능성이 있다. 또한 본 연구에서 제안한 현재 수준의 프로토콜은 데이터 버퍼가 연속된 주소에 존재할 것을 가정하고 있으므로 프로토콜과의 호환성에도 문제가 발생한다. 게다가 스토리지 인터페이스, 즉 공유 메모리 버퍼 영역은 일반적인 용도로는 할당해서는 안되는 공간이기 때문에 같은 인터리빙 그룹에 포함되어 있는 일반적인 DRAM 용량 조차도 인터리빙되어 있는 스토리지 인터페이스로 인해 전혀 사용할수가 없는 공간이 되는 문제가 발생한다. 이와 같이 스토리지 인터페이스가 필요로 하는 주소 공간 이상의 크기를 인터리빙 단위의 최소 크기로 설정할 수 있도록 지원할 필요가 있다.

제안된 스토리지 인터페이스가 인터리빙에 참여하지 않는 경우를 검토해보자. 먼저 첫 번째 경우는 호스트 시스템의 모든 DRAM 장치가 채널, 랭크, 뱅크 관점에서 완벽하게 균등하고 동일하게 연결된 경우다. 이 상황에서는 DRAM의 전체 용량을 호스트 시스템이 보장하는 어떠한 방식으로든 인터리빙시키고, 나머지 주소 공간에 스토리지 인터페이스를 매핑하면 된다. 두번째 경우는 DRAM이 비대칭적으로 연결된 경우, 예를 들면 각 DRAM 채널에 다른 용량의 DRAM이 연결된 경우를 생각해볼 수 있다. 여기서는 DRAM 용량의 대부분을 성능을 위해 인터리빙 방식으로 매핑하고, 인터리빙에 참여하지 않는 남은 용량을 성능을 위해 연속적인 주소 공간에 매핑시키고, 스토리지 인터페이스를 그 외의 영역에 매핑시키면 된다.

이와 같이 호스트 시스템의 DRAM 주소 공간 매핑을 보다 유연하게

설정할 수 있어야 본 연구에서 제안한 스토리지와 스토리지 인터페이스를 효과적으로 적용할 수 있다. 그러나 연구 과정에서 사용한 모바일 CPU는 DRAM의 인터리빙 단위를 조절할 수는 있으나 모든 DRAM이 인터리빙에 참여해야 하는 제약이 있었다. 따라서 인터리빙 단위를 크게 설정함으로써 스토리지 인터페이스가 DRAM과 인터리빙 되지 않는 것과 유사한 효과를 얻을 수 있었다. 단, 위에서 살펴본 것과 같이 주소 공간이 크게 낭비되는 문제가 있으므로 (512 MB의 주소 공간 중 실제로 수 백 KB 만 사용), 향후 모바일 CPU에서는 이에 대한 개선이 필요하다.

6.3.3 DRAM 인터페이스와의 물리적 접속

본 연구의 제안에 의하면 스토리지 장치 컨트롤러가 완벽한 DRAM 인터페이스를 준비해야 한다. 이 요구사항은 스토리지 장치 컨트롤러의 단가에 영향을 주는데, 왜냐하면 DRAM 인터페이스는 일반적인 스토리지 장치 인터페이스에 비해 많은 신호 핀을 사용하기 때문이다. 예를 들면 eMMC는 12개의 핀을 사용하고, UFS 2.0 은 7개 (1 레인) 또는 11개 (2 레인)의 핀을 사용한다. 반면에 LPDDR3 인터페이스는 38개 (x16) 또는 최대 60개 (x32)의 핀을 사용한다. 이와 같이 DRAM 인터페이스는 비교적 많은 핀을 사용하므로 제안된 스토리지 장치 인터페이스를 지원하는 스토리지 장치 컨트롤러가 가격 경쟁력을 확보하기 위해서는 DRAM 인터페이스의 핀 수를 감소시킬 필요가 있다.

현재의 DRAM 인터페이스 표준은 패러랠 방식을 사용한다. 일반적으로 패러랠 방식은 한 클록에 다수의 신호를 동시에 전송함으로써 높은 대역폭과 짧은 레이턴시를 얻을 수 있다. 그러나 패러랠 방식은 클록 스큐 (clock skew)와 크로스 톡 (cross talk)으로 인해 동작 클록을 고속으로 향상시키기 어렵다. 반면 시리얼 방식은 한 클록에 한 비트의 데이터만

전송하기 때문에 한 클록에 전송되는 데이터는 패러랠 방식보다 적다. 하지만 클록 스큐와 크로스 톡과 같은 문제가 없기 때문에 클록을 고속으로 향상시키기 용이하므로 고속 클록을 이용한 성능 향상이 가능하다 [47]. 또한 인터페이스에 다수의 핀이 필요한 패러랠 방식에 비해 시리얼 방식은 소수의 핀을 사용하므로 인터커넥트 비용을 감소시킬 수 있다. 이러한 시리얼 방식의 장점 때문에 스토리지 인터페이스는 SATA, SAS 로, 로컬 버스는 PCIe 로, 비디오/오디오 인터페이스는 HDMI와 같은 시리얼 방식으로 발전해왔다.

DRAM 인터페이스가 시리얼 방식으로 변경되면 현재보다 더 적은 I/O 핀을 사용하면서도 전송속도를 향상시킬 수 있을 것으로 기대된다. 본 논문에서 제안한 스토리지 인터페이스 또한 DRAM 인터페이스 기반이므로 동일한 효과를 얻을 수 있을 것으로 기대된다. 따라서 DRAM 인터페이스가 시리얼 방식으로 변경되면 제안한 스토리지 인터페이스 이용시 기본적으로 얻는 장점인 호스트 시스템의 스토리지 인터페이스 컨트롤러 위한 로직, I/O 핀 감소 외에도 전송속도가 향상되는 장점을 얻을 수 있다.

시리얼 방식의 또다른 장점은 본질적으로 가변 레이턴시를 지원한다는 점이다. 현재의 DRAM 인터페이스는 (변경 가능한) 고정된 레이턴시를 기반으로 동작하기 때문에 레이턴시가 다른 메모리 소자를 사용하는데 큰 제약이 있다. 최근들어 활발하게 연구가 진행되고 있는 SCM (storage class memroy)은 읽기 레이턴시는 DRAM과 유사하고 쓰기 레이턴시는 DRAM 보다 느린 특성이 있다. 그러나 현재의 DRAM 인터페이스는 사전에 지정된 고정된 레이턴시만 지원하므로 DRAM 인터페이스를 통해서는 SCM을 사용할 수 없다. 이 문제를 해결하기 위해 하나의 DRAM 인터페이스 컨트롤러가 DRAM을 포함한 이종의 메모리를 지원하는 방법에 대한 연구가 진행되었다 [48]. DRAM 인터페이스 수준에서 가변 레이턴시를

지원하는 경우 본 논문의 방식과 같이 별도의 소프트웨어 수준 프로토콜을 사용하지 않고 이종의 메모리를 사용할 수 있다.

6.3.4 메모리 트랜잭션 순서 보장

3.3.2에서 논의된 것과 같이 쓰기 명령이 제출되기 전에 반드시 쓰기 데이터가 공유 메모리 버퍼에 쓰여져 있어야 한다. 그러나 스토리지 디바이스 드라이버가 쓰기 데이터를 공유 메모리 버퍼로 전송하는 작업이 쓰기 명령 제출 이전에 수행되도록 작성되더라도 실제로는 두 작업에 대한 메모리 트랜잭션은 L1 (level 1), L2 (level 2) 캐시 등으로 인해 순서가 바뀔 수 있다. 캐시로 인한 메모리 트랜잭션의 순서가 뒤바뀌는 것을 막기 위해 Intel x86 아키텍쳐에서는 `clflush` 인스트럭션을 제공하는데, 이것은 캐시의 내용을 메모리로 플러시한다 [49]. 따라서 쓰기 데이터를 스토리지 인터페이스의 공유 메모리 버퍼로 전송하고 `clflush` 인스트럭션 사용 후 쓰기 명령을 제출하면 적어도 캐시에서 쓰기 데이터가 모두 플러시 된 후 쓰기 명령이 제출되는 것을 보장할 수 있다.

그러나 메모리 트랜잭션의 순서는 호스트 시스템의 메모리 컨트롤러에서도 변경될 수 있다. 일반적인 메모리 컨트롤러는 성능 향상을 위해서 버퍼링 (buffering)이나 메모리 트랜잭션의 리오더링 (reordering) 기법을 적극적으로 활용하기 때문이다 [10]. 따라서 L1, L2 캐시 수준에서의 순서가 보장되더라도 메모리 컨트롤러의 리오더링에 의해 본 연구에서 제안한 호스트 인터페이스에는 쓰기 명령을 제출하는 메모리 트랜잭션이 쓰기 데이터를 전송하는 메모리 트랜잭션보다 먼저 수행될 수 있다. 이것은 본 연구 뿐 아니라 비휘발성을 특성을 가진 SCM이나 NVDIMM-N 장치에도 동일하게 적용되는 문제이다.

이 문제는 시스템 아키텍쳐 수준에서 해결되어야 하는 문제이다. In-

tel x86 아키텍쳐에서는 이 문제를 해결하기 위해 `pcommit` 인스트럭션을 제안하였다 [49]. 이 인스트럭션의 실행이 완료되면 호스트 시스템 메모리 컨트롤러의 버퍼에 있는 트랜잭션이 메모리로 모두 플러시된다. 따라서 쓰기 데이터 전송 후 `clfush` 인스트럭션과 `pcommit` 인스트럭션을 실행하면 공유 메모리 버퍼에 쓰기 데이터가 전송되었음이 보장되고, 그 후 쓰기 명령을 제출하면 쓰기 명령의 무결성을 보장할 수 있다. 그러나 빈번한 캐시 플러시 작업은 시스템 성능을 저하시키므로 캐시 플러시는 가급적 부득이한 경우에만 수행하고, 캐시의 일부 내용만을 플러시하는 `clflushopt` 인스트럭션을 활용하여 성능 저하를 최소화할 수 있다.

본 연구의 구현에 사용된 호스트 시스템의 CPU는 ARM Cortex A15 이다 [50]. 해당 CPU는 아직 `pcommit`과 같이 메모리 컨트롤러의 버퍼를 플러시하는 인스트럭션을 제공하지 않는다. 따라서 구현 과정에서는 캐시 플러시 인스트럭션만을 사용했으나 여기서 논의된 메모리 트랜잭션의 리오더링으로 인한 문제가 발생하지는 않았다. 그러나 향후 해당 CPU가 주로 사용되되는 모바일 환경에서도 본 연구의 스토리지 인터페이스나 SCM 등이 활용되기 위해서는 메모리 컨트롤러 수준에서 트랜잭션 순서를 보장할 수 있는 기법이 필요할 것으로 예상된다.

6.3.5 부팅 지원

하나의 스토리지만으로 컴퓨터 시스템을 운영하기 위해서는 해당 스토리지를 이용해서 운영체제로 부팅할 수 있는 기능이 필수적이다. 기존의 컴퓨터 시스템의 부팅 과정에서는 먼저 시스템 초기화 소프트웨어가 실행되고, 이후 운영체제로의 부팅이 진행된다. 그러나 본 논문에서 제안한 스토리지에서는 부팅이 불가능하므로 이 문제를 해결하기 위해 다음과 같은 지원이 필요하다.

먼저 시스템 초기화 소프트웨어가 제안된 스토리지 (인터페이스)에 대한 디바이스 드라이버를 포함함으로써 운영체제의 부팅 이미지를 읽어들일 수 있어야 한다. 일반적인 컴퓨터 시스템에서는 전원이 인가되면 시스템 초기화와 부팅을 담당하는 소프트웨어가 실행된다. 이 소프트웨어는 데스크탑/서버 시스템에서는 바이오스 (BIOS: basic input output system), 모바일 시스템에서는 부트 로더 (boot loader)라고 불린다. 시스템 초기화 과정은 일반적으로 POST (power on self test) 라고 불리며 [51], CPU, 메인 메모리 뿐 아니라 키보드, 비디오 디스플레이 등의 주변기기에 대한 인식과 초기화를 수행한다. 초기화가 완료되면 스토리지에서 운영체제의 부팅 이미지를 메인 메모리에 읽어들인 후 실행한다. 이와 같이 시스템 초기화 소프트웨어가 스토리지 인터페이스를 위한 디바이스 드라이버를 포함해야 해당 스토리지 인터페이스를 사용하는 스토리지로부터 부팅할 수 있다.

그런데 모바일 시스템은 시스템 리소스의 제약으로 인해 시스템 초기화 소프트웨어의 초기 로딩과 실행이 데스크탑/서버 시스템보다 더 복잡하며, 이로 인해 부팅을 위해 스토리지의 특별한 지원이 필요하다. 데스크탑/서버 시스템의 시스템 초기화 소프트웨어인 바이오스는 별도의 초기화 없이 사용 가능하며 XIP (execution in place)가 가능한 EEPROM (electrically erasable programable read only memory)에 저장되므로, 바이오스의 코드를 EEPROM에 저장된 상태에서 바로 실행할 수 있다. 모바일 시스템에서도 같은 방식으로 부트 로더를 별도의 EEPROM에 저장할 수 있으나, 최신 모바일 시스템에서는 공간 절약과 원가 절감을 위해 부트 로더를 별도의 EEPROM 대신 스토리지에 저장한다. 여기서 발생하는 문제는 다음과 같다. 부트 로더 자체가 시스템의 초기화를 담당하는 소프트웨어이고, 부트 로더가 시스템의 초기화를 완료하고 스토리지 디바이스

드라이버를 로딩해야 스토리지에 대한 I/O를 수행할 수 있기 때문에, 부트 로더가 스토리지에서 로딩되어 실행되기 전에는 스토리지에서 부트 로더를 읽어들일 수 없다는 점이다. 모바일 시스템에서는 이 문제를 해결하기 위해서 부트 로더를 읽어들이는데 특화된 매우 간단한 방법을 사용한다. eMMC 인터페이스의 경우 호스트 시스템이 CMD 핀을 low 상태로 74 클록 사이클이상 유지하면 eMMC 장치는 자동으로 사전에 저장된 부트 로더 이미지를 호스트로 전송한다 [4]. 이와 같은 방식은 *boot mode*라고 불리며, UFS 인터페이스에서도 지원하고 있다 [5]. 해당 방식을 이용하여 부트 로더 데이터를 로딩하는 방법은 코드로 매우 간단하게 구현할 수 있으므로 CPU 에 내장되어 전원이 인가될 때 자동으로 실행되는 부트 ROM 코드에 포함되어 사용되고 있다.

위와 같은 *boot mode*를 제안된 스토리지에 적용하는데는 다음과 같은 어려운 점이 있다. 일반적으로 DRAM 컨트롤러는 부트 로더에 의해 초기화되는데, 제안된 스토리지는 DRAM 인터페이스를 기반으로 하고 있으므로, DRAM 컨트롤러가 초기화되기 전에는 스토리지에서 부트 로더를 읽어오기 어렵다. 따라서 DRAM 컨트롤러가 초기화 되기 전에 eMMC 의 *boot mode*와 같은 간단한 방법을 통해 제안된 스토리지에서 부트 로더를 읽어올 수 있는 방법이 필요하다. 이를 위해 부트 로더 전용의 사이드 밴드 시그널을 고려해 볼 수 있으나, 기존 DRAM 인터페이스 사양에 해당 시그널을 위한 수정이 필요할 수 있다. 또한 이 방법이 구비되더라도 스토리지 인터페이스가 어느 주소에 매핑되어 있는지 알 수 있어야 한다. 다행히도 모바일 시스템의 경우는 데스크탑/서버 시스템과 달리 제조시에 모든 사양이 결정되고 이후 부품이 변경되는 경우는 극히 제한적이므로 제조사가 사용한 스토리지 인터페이스 설정을 그대로 반영하더라도 문제가 없을 것으로 예상된다.

6.3.6 고성능 범용 DMA 지원

본 논문에서 제안한 스토리지 인터페이스의 성능은 실질적으로 호스트 시스템의 메모리와 (`bio` 버퍼)와 스토리지 인터페이스의 공유 버퍼 메모리간의 데이터 전송 성능에 의해 제한된다. DRAM 인터페이스의 수십 GB/s 에 달하는 대역폭을 효과적으로 활용하기 위해서는 고성능 데이터 이동 기법이 필수적이다. 그러나 구현 과정에서 사용된 CPU에 내장된 DMA 장치의 경우는 측정된 최대 속도가 약 450 MS/s 에 불과하여 스토리지 인터페이스의 최대 속도가 그 이하로 제한되었다. 따라서 본 논문 제안과 같이 호스트 시스템에 별도의 스토리지 인터페이스 컨트롤러를 사용하지 않는 경우의 성능을 최대로 끌어내기 위해서는 DRAM 버스에서 DRAM 버스의 대역폭을 최대로 활용할 수 있는 범용 DMA 장치가 필요하다. 또한 컴퓨터 시스템이 처리하는 데이터의 양이 증가함에 따라 대량의 데이터를 전송해야 하는 상황이 점점 더 자주 발생할 것으로 보인다. 이러한 대량의 데이터 전송을 효과적으로 처리할 수 있도록 특정 서버 CPU의 경우는 범용의 고성능 DMA 장치를 제공함으로써 대량의 데이터에 대한 고속 전송을 지원하고 있다 [52]. DMA와 같은 별도의 하드웨어 장치가 아닌 대량의 데이터 이동을 위한 CPU 인스트럭션이 제공되는 경우도 있으므로 DMA 사용이 불가능한 경우는 해당 인스트럭션을 활용하는 것도 가능하다.

제 7 장

결론 및 향후 연구

7.1 결론

지금까지 본 논문에서 제안한 DRAM 인터페이스 기반의 스토리지 아키텍쳐의 설계, 구현, 평가를 살펴보았다. 먼저 DRAM 인터페이스를 기반으로 스토리지 인터페이스를 설계하였다. 스토리지 인터페이스는 공유 메모리 버퍼와 이를 기반으로 동작하는 스토리지 프로토콜로 구성된다. 그리고 이 인터페이스를 기반으로 동작하는 스토리지 장치를 설계하였다. 스토리지 장치는 스토리지 컨트롤러와 펌웨어로 구성된다. 또한 제안된 스토리지 장치를 위한 호스트 시스템을 설계한다. 호스트 시스템은 호스트 플랫폼 하드웨어, 부트 로더, 디바이스 드라이버로 구성된다. 그 후 스토리지 장치와 호스트 시스템을 구현하고 통합하여 하나의 완전한 Android 시스템을 구축함으로써 제안의 타당성을 검증하였다. 실제 시스템에서의 성능 평가 결과 여러 가지 약점에도 불구하고 최신의 상용 UFS 2.0 장치보다 빠른 순차 읽기 성능을 보였다. 따라서 본 논문에서 제안한 스토리지 아키텍쳐의 타당성이 검증되었다고 할 수 있다.

7.2 향후 연구

지금까지 DRAM 호스트 인터페이스를 사용하는 스토리지의 설계, 구현, 그리고 평가를 통해 제안된 스토리지 아키텍쳐의 타당성을 확인했

다. 그러나 제안된 스토리지를 본격적으로 사용할 때의 전체 시스템에 대한 영향성에 대한 추가 연구가 필요할 것으로 예상된다. 제안된 스토리지를 사용할 경우 DRAM 버스 대역폭의 사용량이 증가하며, DRAM의 인터리빙이 제약되기 때문에 전체 시스템 성능에 영향을 줄 수 있기 때문이다. 또한 제안된 스토리지 인터페이스는 기존의 스토리지 인터페이스에 비해 낮은 레이턴시를 제공하기 때문에, 낮은 레이턴시를 제공하는 차세대 메모리와 결합하면 고성능 차세대 메모리를 더 효과적으로 활용할 수 있을 것으로 기대된다. 따라서 제안된 인터페이스와 차세대 메모리를 활용하는 고성능 스토리지 아키텍쳐에 대한 연구도 필요하다.

또한 스토리지 인터페이스가 호스트 시스템의 주소 공간에 매핑되는 버퍼라는 특성을 적극적으로 활용하기 위한 방안에 대한 연구도 필요하다. 해당 버퍼 영역은 호스트 시스템에서 DRAM과 동일한 용도로 사용할 수 있기 때문에 용량이 작은 NVDIMM-N과 유사하게 활용할 수 있다. 용량이 작기 때문에 대용량 데이터를 저장하는 용도로는 활용이 불가능하지만, 용량이 작은 임시 데이터를 스토리지에서 읽어서 잠시 참조한 후 버퍼에서 버리는 방식으로 사용하면 해당 버퍼에서 호스트 시스템의 메모리로의 데이터 전송 시간과 사용되는 메모리 버스 대역폭을 줄일 수 있다. 업데이트가 빈번히 발생하는 저널링/로깅 버퍼로 활용하는 방안에 대한 연구도 필요할 것으로 보인다.

궁극적으로는 메모리와 스토리지의 통합된 인터페이스에 및 프로토콜에 대한 연구가 필요하다. 앞에서 살펴본 것과 같이 현재는 메모리와 스토리지 인터페이스가 별도로 분리되어 있다. 하지만 고성능의 차세대 메모리 소자가 현실화되고 있어 메모리와 스토리지의 성능 차이가 점점 감소하고 있다. 따라서 메모리와 스토리지를 통합하여 사용할 수 있는 인터페이스 및 프로토콜에 대한 연구가 반드시 필요하다.

마지막으로 이종 메모리로 단일 시스템에서 이종 메모리를 효과적으로 활용할 수 있는 바안에 대한 연구가 필요하다. 현재의 컴퓨터 시스템은 주로 빠르고 휘발성인 DRAM, 느리고 비휘발성인 스토리지의 두 가지 요소를 위주로 최적화가 되어 있다. 그러나 차세대 메모리의 경우 기존의 저장 소자와는 다른 휘발성, 속도, 전력 소모, 가격 특성을 가질 수 있기 때문에 이를 고려하여 메모리 시스템을 최적으로 구성하고 운영할 수 있는 방안에 대한 연구가 필요하다. 또한 이러한 차세대 메모리를 사용하면 현재보다 더 큰 메모리를 확보할 수 있게 되는데, 이러한 대용량 메모리 시스템을 위한 가상 메모리 운영 방안 연구도 반드시 필요하다.

참고 문헌

[1] S. Seo, Y. Cho, Y. Yoo, O. Bae, J. Park, H. Nam, S. Lee, Y. Lee, S. Chae, M. Kwon, *et al.*, "Design and implementation of a mobile storage leveraging the dram interface," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 531–542, IEEE, 2016.

[2] "Serial attached scsi." `https://en.wikipedia.org/wiki/Serial_Attached_SCSI`.

[3] NVM Express, Inc., "Nvm express revision 1.2." `http://nvmexpress.org/wp-content/uploads/NVM_Express_1_2_Gold_20141209.pdf`, November 2014.

[4] JEDEC, "Embedded multi-media card electrical standard (5.1), jesd84-b50.1." `http://www.jedec.org/sites/default/files/docs/JESD84-B51.pdf`, February 2015.

[5] JEDEC, "Universal flash storage (ufs), version 2.0, jesd220b." `https://www.jedec.org/sites/default/files/docs/JESD220B.pdf`, September 2013.

[6] JEDEC, "Universal flash storage unified memory extension,version 1.0, jesd220-1." `https://www.jedec.org/sites/default/files/docs/JESD220-1.pdf`, September 2013.

[7] K. Watanabe, K. Yoshii, N. Kondo, K. Maeda, T. Fujisawa, J. Wadatsumi, D. Miyashita, S. Kousai, Y. Unekawa, S. Fujii, *et al.*, "19.3 66.3 kiops-random-read 690mb/s-sequential-read universal flash storage device controller with unified memory extension," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pp. 330–331, IEEE, 2014.

[8] JEDEC, "Ddr4 sdram, jesd79-4." http://www.jedec.org/standards-documents/docs/jesd79-4, September 2012.

[9] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[10] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *ACM SIGARCH Computer Architecture News*, vol. 36, pp. 63–74, IEEE Computer Society, 2008.

[11] "Jffs2." https://en.wikipedia.org/wiki/JFFS2.

[12] "Yaffs — a flash file system for embedded use." http://www.yaffs.net.

[13] "Yaffs." https://en.wikipedia.org/wiki/YAFFS.

[14] "Ubifs." https://en.wikipedia.org/wiki/UBIFS.

[15] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, (Santa Clara, CA), pp. 273–286, USENIX Association, Feb. 2015.

[16] "F2fs." https://en.wikipedia.org/wiki/F2FS.

[17] "Nvdimm." https://en.wikipedia.org/wiki/NVDIMM.

[18] J. Chang and A. Sainio, "NVDIMM-N Cookbook." http://www.snia.org/sites/default/orig/FMS2015/Chang-Sainio_NVDIMM_Cookbook.pdf.

[19] JEDEC, "JEDEC Announces Support for NVDIMM Hybrid Memory Modules." http://www.jedec.org/news/pressreleases/jedec-announces-support-nvdimm-hybrid-memory-modules.

[20] Agigatech, "Agigaram ddr3 nvdimm." http://www.agigatech.com/ddr3.php.

[21] I. Netlist, "Nvvault ddr3 nvdimm." http://www.netlist.com/vault-memory-storage/nvvault-ddr3.

[22] V. Technology, "Non-volatile memory." http://www.vikingtechnology.com/nvdimm-technology.

[23] S. Hynix, "16gb nvdimm." https://www.skhynix.com/eng/popup/prdNVlDIMM.jsp.

[24] J. McFarland, "Memory channel storage (mcs) demystified." http://www.snia.org/sites/default/files/Memory%20Channel%20Storage%20Demystified%20-%203_10_14.pdf.

[25] R. Callaghan, "ULLtraDIMM SSD Overview." http://www.snia.org/sites/default/files/SanDisk%20ULLtraDIMM_0.pdf, June 2014.

[26] "Memory Reference Code." https://en.wikipedia.org/wiki/Memory_Reference_Code.

[27] "E820." https://en.wikipedia.org/wiki/E820.

[28] "Memory type range register." https://en.wikipedia.org/wiki/Memory_type_range_register.

[29] "Memory Training, Testing, and Margining." http://blog.asset-intertech.com/test_data_out/2014/11/memory-training-testing-and-margining.html, November 2014.

[30] "Advanced Configuration and Power Interface." https://en.wikipedia.org/wiki/Advanced_Configuration_and_Power_Interface.

[31] "Netlist announces strategic partnership with samsung for new storage class memory." http://www.netlist.com/investors/financial-news/press-release-details/2015/Netlist-Announces-Strategic-Partnership-With-Samsung-For-New-Storage-Class-Memory/default.aspx, November 2015.

[32] Intel, "Chip shot: Intel unveils intel optane technology based on 3d xpoint." http://newsroom.intel.com/community/intel_newsroom/blog/2015/08/19/chip-shot-intel-unveils-intel-optane-technology-based-on-3d-xpoint, August 2015.

[33] Unified EFI, Inc., "Advanced configuration and power interface specification 6.0." http://www.uefi.org/sites/default/files/resources/ACPI_6.0.pdf, april 2015.

[34] "JEDEC Byte-Addressable Energy-Backed Interface NVDIMM Device-Specific Method." https://msdn.microsoft.com/en-us/library/windows/hardware/mt604728(v=vs.85).aspx.

[35] "Storage Class Memory Support in the Windows Operating System." http://www.snia.org/sites/default/files/SDC15_presentations/file_sys/NealChristiansen_SCM_on_Windows.pdf.

[36] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *FAST*, vol. 12, pp. 3–3, 2012.

[37] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer, "Architecture support for improving bulk memory copying and initialization performance," in *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pp. 169–180, IEEE, 2009.

[38] Intel, *The IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*. 2003.

[39] Samsung Electronics, "Exynos 5250 user's manual." http://www.samsung.com/global/business/semiconductor/file/product/Exynos_5_Dual_User_Manaul_Public_REV100-0.pdf.

[40] YIC System, "SMDKC520." http://www.yicsystem.com/products/smdk-board/smdk-c520.

[41] DENX Software Engineering, "The universal boot loader." http://www.denx.de/wiki/U-Boot/WebHome.

[42] D. P. Bovet and M. Cesati, *Understanding the Linux kernel*. " O'Reilly Media, Inc.", 2005.

[43] R. Love, *Linux kernel development*. Pearson Education, 2010.

[44] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers*. " O'Reilly Media, Inc.", 2005.

[45] D. E. Knuth, *Art of Computer Programming Volume 1: Fundamanetal Algorithms*. Addison-Wesley Publishing Company, 1972.

[46] "Iozone filesystem benchmark." http://www.iozone.org.

[47] "Serial communication." https://en.wikipedia.org/wiki/Serial_communication.

[48] T. J. Ham, B. K. Chelepalli, N. Xue, and B. C. Lee, "Disintegrated control for energy-efficient and heterogeneous memory systems," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 424–435, IEEE, 2013.

[49] Intel, "Intel architecture instruction set extensions programming reference, 319433-023." https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf, August 2015.

[50] "Cortex-A15 Processor." http://www.arm.com/products/processors/cortex-a/cortex-a15.php.

[51] "Bios." https://en.wikipedia.org/wiki/BIOS.

[52] Intel, "Intel quickdata technology software guide for linux." http://www.intel.com/content/dam/doc/white-paper/quickdata-technology-software-guide-for-linux-paper.pdf, May 2008.

Abstract

Design and Implementation of A High Performance Storage Leveraging the DRAM Host Interface

Sungyong Seo
Department of Electrical Engineering & Computer Science
The Graduate School
Seoul National University

Storage is a key factor that determines the overall performance of a computer system. In the era of big data, the demand for high performance computer systems has been ever increasing. High performance storage is also needed in order to construct a high performance computer system. The performance of a storage has increased dramatically with the adoption of NAND flash memory.

A storage is connected with a host system via a storage interface. The storage interface has evolved in order to fully exploit the performance of a storage. Its performance will evolve as storage advances. However, we already have faster interface in current computer system: the DRAM inter-

face. It provides up to 25.6 GB/s in case of latest DDR4 specification. Since the protocols for storage and DRAM are not compatible, we cannot exploit the DRAM interface as a storage interface as is.

In this work, a new storage protocol is proposed in order to turn the DRAM interface to a storage interface. It runs on top of the DRAM interface. This protocol builds on a small host interface buffer structure mapped to the host system's memory space. Given the protocol, a design of storage controller and firmware is proposed. The storage controller natively supports the DRAM (LPDDR3) interface. Also a new host platform including both hardware and software is proposed for the proposed storage since the storage cannot be connected with conventional computer systems.

Finally the feasibility of this work is proved by constructing a full Android system running on the developed storage and platform. Evaluation result shows that the proposed storage architecture has very low protocol handling overheads and compares favorably to a latest commercial UFS 2.0 storage.

Keywords : Mobile storage, DRAM interface storage, NVDIMM

Student Number : 2004-23577