creative commons
COMMONS DEED

저작자표시-비영리-동일조건변경허락 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

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

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

동일조건변경허락. 귀하가 이 저작물을 개작, 변형 또는 가공했을 경우에는, 이 저작물과 동일한 이용허락조건하에서만 배포할 수 있습니다.

- 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

creative commons
COMMONS DEED

저작자표시-비영리-동일조건변경허락 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

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

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

동일조건변경허락. 귀하가 이 저작물을 개작, 변형 또는 가공했을 경우에는, 이 저작물과 동일한 이용허락조건하에서만 배포할 수 있습니다.

- 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

병렬화된 플래시 메모리 저장 장치에서 불량 블록 관리의 정확성 보장 및 성능 최적화 기법

Performance optimization of a bad block management scheme with guaranteed correctness in parallelized flash memory storage devices

2013년 8월

서울대학교 대학원

컴퓨터공학과

김 홍 석

병렬화된 플래시 메모리 저장 장치에서 불량 블록 관리의 정확성 보장 및 성능 최적화 기법

Performance optimization of a bad block management scheme with guaranteed correctness in parallelized flash memory storage devices

지도교수 민 상 렬

이 논문을 공학박사 학위논문으로 제출함

2013년 8월

서울대학교 대학원

컴퓨터공학과

김 홍 석

김홍석의 공학박사 학위논문을 인준함

2013년 8월

위 원 장 _____ (인)

부 위 원 장 _____ (인)

위 원 _____ (인)

위 원 _____ (인)

위 원 _____ (인)

초 록

다수의 연산들이 병렬적으로 실행되는 플래시 메모리 저장 장치에서 불량 블록으로 인한 플래시 메모리 오류가 발생하면 연산 간의 의존 관계로 인해 FTL과 같은 플래시 메모리 관리 소프트웨어의 동작 정확성을 보장하기 어렵다. 더욱이 실용적인 플래시 메모리 저장 장치에서는 최적화된 성능을 제공하기 위해 플래시 메모리 오류로 인한 불량 블록 관리 부담을 최소화하는 것이 필수적이다.

본 논문의 불량 블록 관리 기법은 다수의 플래시 메모리 연산들이 병렬적으로 실행되는 환경에서 불량 블록 처리 작업과 관련된 연산 간의 의존성 문제를 해결한다. 제시하는 기법은 동작의 정확성을 보장하면서도 불량 블록 처리 중에도 일반 연산이 실행되는 것을 허용하여 상위 계층 플래시 메모리 소프트웨어가 생성하는 다수의 플래시 메모리 연산들에 대한 처리 지연을 방지한다. 또한 상위 계층 요청 처리 과정에서 필수적인 주소 변환 과정을 재사상 블록 필터와 사상 정보 캐시를 활용하여 가속화함으로써 불량 블록 주소 관리 기법의 성능을 최적화한다. 구현과 관련하여 플래시 메모리 저장 장치의 성능과 자원 요구 사항에 따라 하드웨어 또는 소프트웨어 구현 모두를 지원하기 때문에 자원이 제약된 소형 저장 장치와 대용량 고성능 저장 장치 모두에 효과적으로 적용될 수 있다.

불량 블록 관리 기법을 소프트웨어 및 하드웨어로 구현하였고, 다수의 플래시 메모리 연산 중에 임의로 동기적/비동기적 오류가 중첩하여 발생하는 시뮬레이션 환경에서 동작을 검증하였다. 또한 병렬 실행 플래시 메모리 제어기가 탑재된 FPGA 개발 보드 상에서 성능을 평가하여 구현

의 유효성과 실효성을 입증하였다. 실험 결과, 적은 수의 플래시 메모리 칩을 사용하는 저장 장치에서는 소프트웨어 구현만으로도 플래시 제어기가 제공하는 대역폭을 95% 이상 활용하였다. 그러나 SSD와 같이 다수의 플래시 메모리를 사용하는 고성능 저장 장치에서는 소프트웨어 개입으로 인한 요청 처리의 지연이 성능 저하 문제를 초래하였다. 하드웨어 지원 구현의 경우, 대용량 저장 장치에서도 플래시 제어기 대역폭의 97% 이상을 활용하였다.

주요어: 플래시 메모리, 불량 블록, 불량 블록 관리 기법, 병렬 처리, 의존성 문제, 하드웨어 가속화, 플래시 변환 계층 (FTL)

학 번: 2005-21375

목 차

제 1 장 서 론 ····· 1

1.1 연구 동기 ····· 1

1.2 연구 내용 ····· 5

1.3 논문의 구성 ····· 7

제 2 장 배경 지식 및 관련 연구 ····· 9

2.1 플래시 메모리 ····· 9

2.2 플래시 메모리의 기본 원리 ····· 10

2.3 플래시 메모리의 신뢰성 특성 ····· 18

2.4 병렬 연산 처리를 위한 플래시 메모리의 내부 구조 ····· 25

2.5 플래시 메모리 기반 저장 장치 구조 ····· 31

2.6 불량 블록 관리 기법 ····· 43

2.6.1 개요 ····· 43

2.6.2 FTL과 하위 계층에 분산된 구현 방식 ····· 46

2.6.3 FTL과 독립적으로 구현된 방식 ····· 50

제 3 장 의존성 문제 및 해결 방안 ····· 60

3.1 불량 블록 관리 기법 (BMS)에 대한 가정 ····· 61

3.2 의존성 문제 ····· 67

3.3 의존성 해결 방안 ····· 71

3.3.1 데이터 의존성 해결 방안 ····· 75

3.3.2 쓰기 순서 의존성 해결 방안 ····· 77

3.3.3 재사상에 의한 의존성 해결 방안 ····· 86
3.3.5 논의 ····· 88

제 4 장 병렬화된 불량 블록 관리 기법 설계 ····· 92
4.1 가정 및 설계 목표 ····· 92
4.2 전체 구조 ····· 95
4.2.1 순차적으로 동작하는 BMS의 구조 ····· 96
4.2.2 병렬화된 BMS의 기본 구조 ····· 97
4.2.3 정확성을 보장하는 병렬화된 BMS 구조 ····· 100
4.3 하드웨어 기반 최적화 설계 ····· 106
4.3.1 하드웨어 지원 설계 ····· 108
4.3.2 효율적인 자료 구조 관리 방안 ····· 112

제 5 장 검증 및 성능 평가 ····· 123
5.1 시뮬레이션 기반 정확성 검증 ····· 123
5.2 성능 평가 ····· 126
5.2.1 실험 환경 ····· 126
5.2.2 성능 평가 ····· 129

제 6 장 결론 및 향후 연구 과제 ····· 136

참고문헌 ····· 140

Abstract ····· 152

그림 목차

그림 1 플래시 메모리 셀의 구조 ····· 10
그림 2 셀의 상태 및 임계 전압 ····· 11
그림 3 F-N 터널링에 의한 상태 전이 ····· 12
그림 4 플래시 메모리 블록의 구조 ····· 13
그림 5 MLC 셀의 상태 ····· 16
그림 6 셀 당 비트 수에 따른 잡음 여유 ····· 17
그림 7 지우기/쓰기 반복에 의한 셀의 마모 과정 ····· 18
그림 8 초창기 플래시 메모리의 내부 구조 ····· 25
그림 9 다중 LUN 구조 ····· 26
그림 10 다중 플레인 구조 ····· 27
그림 11 캐시 레지스터 구조 ····· 29
그림 12 플래시 메모리 저장 장치의 기본 구조 ····· 32
그림 13 플래시 메모리 칩의 연결 방식 ····· 40
그림 14 플래시 메모리 오류의 분류 ····· 55
그림 15 페이지 동작 모델 ····· 57
그림 16 X-BMS 및 RMTD에서의 블록 수준 사상 방식 ····· 63
그림 17 FTL요청 처리 과정 ····· 65
그림 18 쓰기 요청 처리 과정 (쓰기 연산이 성공한 경우) ····· 78
그림 19 순차적인 쓰기 요청 처리 과정 (쓰기 연산이 성공한 경우) · 78
그림 20 중첩된 쓰기 요청 처리 (쓰기 연산이 성공한 경우) ····· 81

그림 21 FTL 쓰기 요청 처리 과정 (쓰기 오류가 발생한 경우) ····· 82
그림 22 재사상으로 인한 쓰기 손실 문제 ····· 83
그림 23 작업 무효화를 통한 쓰기 손실 해결 ····· 85
그림 24 순차적으로 동작하는 BMS 구조 ····· 96
그림 25 병렬화된 BMS 기본 구조 ····· 98
그림 26 정확성 보장을 위한 병렬화된 BMS 구조 ····· 103
그림 27 블룸 필터 동작 예시 ····· 115
그림 28 블룸 필터 긍정 오류 확률 ····· 116
그림 29 재사상 블록 필터의 활용 ····· 118
그림 30 사상 정보 캐시의 활용 ····· 120
그림 31 시뮬레이션 기반 검증 환경 ····· 123
그림 32 평가 플랫폼 보드 ····· 126
그림 33 다양한 NAND 모듈 ····· 126
그림 34 플래시 메모리 매개 변수 ····· 127
그림 35 FPGA 자원 사용량 ····· 128
그림 36 실험 환경 구성 ····· 129
그림 37 버스 및 칩 개수에 따른 요청 처리율 (정상 동작 상황) ····· 130
그림 38 응답 시간 분포 (정상 동작 상황) ····· 133
그림 39 버스 및 칩 개수에 따른 요청 처리율 (오류 상황 1) ····· 134
그림 40 버스 및 칩 개수에 따른 요청 처리율 (오류 상황 2) ····· 135

제 1 장 서 론

1.1 연구 동기

플래시 메모리는 전원이 공급이 중단된 상태에서도 저장된 데이터가 유지되는 비휘발성 반도체 메모리로서 크기가 작으면서도 빠른 임의 접근 속도, 저전력, 충격과 진동에 대한 내구성 등의 장점으로 인하여 스마트폰, 태블릿 PC (Tablet PC), PDA, USB 드라이브와 같은 이동형 저장 매체로서 널리 사용되고 있다. 또한 최근에는 제조 공정의 세밀화를 기반으로 하나의 셀에 여러 비트의 정보를 저장하는 MLC (Multi-Level Cell) 기술 및 기판 위에 복수의 플래시 메모리 칩을 적층하는 MCP (Multi-Chip Package) 기술이 활용되어 플래시 메모리의 저장 용량은 빠른 속도로 증가되고 있다.

플래시 메모리는 단일 PCB (Printed Circuit Board) 상에 다수의 칩을 장착하여 칩 수준의 병렬 처리가 가능한데 이러한 특성으로 인하여 성능 역시 급격한 향상을 보이고 있다. 이에 따라 플래시 메모리는 용량 대비 가격과 함께 성능 측면에서도 경쟁력을 확보하게 되었으며, 서버용 캐시, SSD (Solid State Drive)와 같은 고성능 대용량 저장 장치 응용에 적용되어 PC나 서버 시스템에서 하드 디스크를 대체하고 있다 [1][2][3].

이러한 다양한 장점과는 별도로 플래시 메모리는 인터페이스 측면에서 심각한 제약이 있다. 일반적인 저장 장치와는 달리 덮어 쓰기 (Overwrite)가 불가능하여 쓰여진 데이터를 갱신하기 위해서는 지우기

연산이 선행되어야 한다. 그런데 지우기 연산은 블록 (Block) 단위로 수행되는 반면 읽기 및 쓰기 연산은 블록을 구성하는 페이지 단위로 수행된다. 이와 같이 지우기 연산의 선행을 수반하는 덮어 쓰기에 대한 제약은 저장 장치의 쓰기 성능을 심각하게 저하시킬 수 있으며, 지우기 연산과 읽기 및 쓰기 연산 단위의 불일치는 플래시 메모리를 관리하는 복잡도를 증가시킨다.

인터페이스에서의 제약과 함께 신뢰성 측면에서도 심각한 문제들이 있다. 플래시 메모리는 정상적으로 쓰여진 데이터라도 읽기 연산 시 비트가 일시적으로 반전되는 오류 (Bit-flipping error)가 발생하는 문제점이 있다. 또한 플래시 메모리는 제조 시 물리적인 결함으로 인하여 연산에 대한 정확한 동작을 보장하지 못하는 불량 블록이 포함될 수 있으며, 더욱이 정상적인 블록도 쓰기와 지우기 연산이 반복됨에 따라 블록을 구성하는 셀이 마모되어 불량 블록이 될 수 있다. 이러한 동작 결함 특성으로 인하여 대부분의 제조사는 정상 블록에 대해서도 허용 가능한 쓰기 및 지우기 반복 횟수를 명시하는데 이를 내구성 (Endurance)이라고 한다. 일반적으로 SLC (Single-Level Cell) 플래시 메모리는 내구성 수치가 100,000번 정도이고 MLC (Multi-Level Cell) 플래시 메모리는 3,000번에서 10,000 정도이다. 이와 같은 플래시 메모리의 신뢰성 문제는 제조 공정이 세밀해짐에 따라 점점 악화되고 있다[4].

이와 같은 다양한 플래시 메모리의 제약 사항을 극복하기 위하여 플래시 변환 계층 (Flash Translation Layer; FTL)[5]이라는 소프트웨어 계층이 사용된다. FTL은 하드 디스크와 유사한 저장 장치 인터페이스를 호스트 시스템에 제공하며 이를 위하여 호스트 시스템 상의 논리 주소를 플래시 메모리 상의 물리 주소로 변환하는 주소 사상 (Mapping) 기능을 수행한다.

이와 함께 성능 향상을 위하여 버퍼 관리 및 칩 수준 또는 작업 수준 병렬 처리를 수행하며, 저장 공간 확보를 위한 쓰레기 수집 (Garbage collection)과 같은 복잡한 기능을 수행한다. 또한 신뢰성 문제를 해결하기 위해 불량 블록 관리, 마모도 평준화 (Wear leveling), 전원 오류 복구 (Crash recovery) 등의 작업 역시 수행한다.

이와 같은 FTL의 기능 및 복잡도는 점점 증가하고 있는데 이는 플래시 메모리의 신뢰성 저하와 함께 성능과 기능 측면에서 사용자의 요구 사항이 다양해지고 있기 때문이다. 따라서 위에서 설명한 많은 기능을 FTL에서 수행한다면 과도한 복잡도로 인해 성능 개선이나 인터페이스 확장과 같은 다양한 요구 사항을 만족시키기 어려우며, 구현한 FTL의 정확성을 검증하는 것도 현실적으로 불가능해진다. 이를 해결하기 위해 계층화된 설계 방법을 통해 FTL의 기능을 분산시키는 방법이 개발되고 있으며, 최근에는 디바이스 드라이버와 같이 독립된 계층에서 불량 블록을 관리하는 방안이 연구되고 있다[9][10].

계층화된 구조에서는 전체 시스템의 성능과 신뢰성을 보장하기 위해서 각 구성 요소가 효율적이며 정확하게 설계되어야 한다. 그러나 독립된 계층에서의 불량 블록 관리 기법에 대한 연구는 미흡한 실정인데 특히 플래시 메모리에서 발생 가능한 모든 오류 상황을 고려한 정확한 불량 블록 처리에 대한 심도 있는 연구가 필요한 상황이다. 이는 불량 블록 처리 과정에서는 연산 오류나 전원 오류와 같은 다양한 종류의 오류가 중첩하여 반복적으로 발생할 수 있는데 이러한 특성으로 인하여 발생 가능한 모든 상황을 고려한 정확성 보장이 어렵기 때문이다. [9]에서는 X-BMS라는 불량 블록 관리 기법을 제안하고 정확성을 정형적으로 검증하였다. 그러나 다수의 플래시 메모리 연산이 병렬적으로 실행되는 상

황을 고려하지 않았기 때문에 상용 저장 장치에 적용되는 경우 심각한 성능 저하를 초래할 수 있다. 만약 X-BMS에서 병렬 처리를 허용하는 경우 연산 간의 의존성 (Dependence) 문제로 인하여 불량 블록 관리의 정확성을 보장하지 못하여 FTL 데이터가 손실되는 심각한 결과를 초래할 수 있다.

병렬 처리와 관련된 정확성 문제 이외에도 기존 연구들에는 실용적인 문제가 있는데, 이는 계층화로 인한 성능 저하가 발생할 수 있다는 것이다. 기존의 불량 블록 관리 기법은 소프트웨어 형태로 구현되며 사상 과정을 통해 불량 블록에 대한 FTL의 요청을 정상 블록에 대한 연산으로 변환한다. 이와 같은 불량 블록 처리를 위한 부가의 소프트웨어 작업은 FTL 요청 처리가 지연되는 상황을 유발하는데 플래시 메모리의 연산 실행 시간이 하드 디스크와 같은 기존 저장 장치에 비해 매우 짧기 때문에 요청 처리 지연으로 인한 성능 저하가 발생할 수 있다. 이러한 성능 저하 문제는 서버용 SSD와 같은 고성능 응용 제품의 경쟁력을 크게 저해할 것이다.

이러한 문제점을 해결하기 위해 병렬 처리 시에도 정확성을 유지하고 성능 저하를 방지하는 불량 블록 관리 기법에 대한 연구가 필요하다. 병렬 처리 상황에서는 연산 간의 의존성 문제가 발생하며, 이는 불량 블록 처리와 결부되어 정확성 측면에서 다양한 문제 상황을 일으킨다. 더욱이 이러한 문제 상황은 불량 블록 관리 기법의 구현 방식에 종속적으로 발생하게 된다. 따라서 구현에 독립적인 방식으로 의존성 문제를 규명하고 이를 해결할 수 있는 효율적인 방안이 수립되어야 한다. 또한 계층화로 유발된 성능 저하를 해결하기 위해 다양한 불량 블록 관리 기법에 적용될 수 있는 최적화 방안이 제시되어야 한다.

1.2 연구 내용

본 논문의 목표는 FTL 요청을 최대한 병렬적으로 처리하는 동시에 연산 간의 의존성 문제를 해결하는 불량 블록 관리 기법을 제시하는 것이다. 제시한 기법은 계층화로 인한 성능 저하 문제를 해결하기 위해 하드웨어로 구현 가능해야 한다. 또한 기존의 불량 블록 기법이 가지고 있는 자료 구조의 비효율성을 보완하는 최적화 방안을 마련하여, 성능 병목 현상을 방지해야 한다.

이를 위하여 기존 불량 블록 관리 기법을 분석하고 불량 블록 처리 중에 발생하는 의존성 문제를 규명한다. 의존성에는 불량 블록 관리와 무관하게 플래시 메모리 저장 장치에서 일반적으로 적용되는 데이터 의존성과 쓰기 순서 제약에 의한 의존성이 있으며, 불량 블록 관리 기법과 연관되어 발생하는 재사상에 의한 의존성이 있다. 이러한 의존성들이 발생하는 모든 경우를 체계적으로 정리한다.

의존성 문제를 효율적으로 해결하기 위해서는 FTL로부터의 요청 특성을 고려하는 것이 필요하다. FTL은 다양한 기능을 수행하기 위해 다수의 모듈로 구성되어 있으며 정확성을 보장하기 위해 각 모듈들은 동기화된 방식으로 동작한다. 따라서 각 모듈들이 생성하는 요청 간에는 데이터 의존성이 발생하지 않거나 극히 드문 경우에만 발생하게 된다. 이와 같은 특성을 고려하여 데이터 의존성은 불량 블록 관리 기법과 독립된 계층에서 해결하도록 하여 구현 복잡도를 완화한다. 그 외의 의존성은 상시적으로 발생하거나 사전에 발생 여부를 파악할 수 없기 때문에 불량 블록 관리 기법과 연관하여 해결한다. 이 때 요청 처리 지연을 최소화할 수 있는 요청 실행 규칙을 마련하도록 한다.

의존성 해결 규칙이 수립되면 이를 바탕으로 성능을 최적화할 수 있는 설계 구조를 제안한다. 다수의 플래시 메모리 칩을 최대한 병렬적으로 활용하는 것은 플래시 메모리 제어기에서 가장 효율적으로 할 수 있기 때문에 불량 블록 관리 기법을 구현하는 계층에서는 플래시 메모리에 충분한 작업 부하 (Workload)를 공급하는 것에 집중하도록 한다. 이러한 원칙에 따라 FTL 요청의 처리 과정을 세부 단계로 나누고 각 단계가 병렬적으로 수행될 수 있는 설계 구조를 제시한다. 또한 하드웨어 기반의 구현을 통해 성능을 더욱 개선할 수 있도록 각 세부 단계에서는 단순한 작업만이 수행되도록 한다. 그리고 재사상 블록 필터와 사상 정보에 대한 캐시를 도입하여 요청 처리 과정에서 병목이 될 수 있는 주소 변환 과정을 최적화한다.

이와 같은 설계 방안을 바탕으로 불량 블록 관리 기법을 구현하고 동기적 / 비동기적 오류가 임의로 중첩되어 발생하는 시뮬레이션 환경에서 구동하여 정확성을 검증한다. 또한 무순서 플래시 메모리 제어기가 탑재된 실제 플랫폼에서 구동하여 불량 블록 관리로 인해 FTL이 체감하는 성능 저하를 분석하고 구현한 불량 블록 관리 기법의 실효성 및 유효성을 평가한다.

본 연구의 의의는 다음과 같다.

(1) 불량 블록 처리 과정에서 발생 가능한 FTL 요청 간의 의존성 문제를 정리하였다. FTL 요청들은 접근하는 블록 및 페이지 주소뿐만 아니라 연산 수행 중에 발생한 불량 블록을 처리하는 과정에서 의존성이 발생할 수 있으며, 가능한 모든 경우를 고려하여 의존성 문제를 체계적으로 규명하였다.

(2) 의존성에 대한 효율적인 해결 방안을 제시하였다. 이를 위해 FTL의 요청 특성을 고찰하여 의존성 종류에 따른 발생 빈도를 예측하였고,

이를 기반으로 구현 복잡도를 줄이고 대부분의 경우에 요청 처리 지연을 회피할 수 있는 실행 규칙을 마련하였다.

(3) 의존성 해결을 위한 실행 규칙을 만족하면서, FTL 요청을 병렬적으로 처리할 수 있는 설계 구조를 제시하였다. 해당 설계에서는 불량 블록 관리 기법을 위한 기능들을 독립적인 모듈에서 수행하도록 하여 병렬 처리 효율을 개선하였으며, 하드웨어 구현을 통한 추가의 성능 향상 방안을 제시하였다. 또한 성능에 병목이 될 수 있는 자료 구조에 대한 최적화 방안을 제시하였다.

(4) 제안한 설계에 따라 불량 블록 관리 기법을 소프트웨어 및 하드웨어로 구현하여 FPGA 기반 개발 보드에서 성능을 비교 평가 하였으며, 시뮬레이션 환경에서 정확성을 검증하였다.

1.3 논문의 구성

본 논문의 구성은 다음과 같다. 제 2장에서는 본 연구 주제에 대한 배경 지식으로서 플래시 메모리의 기본 원리, 신뢰성 특성, 병렬 실행을 위한 구조를 살펴본다. 또한 플래시 메모리 저장 장치의 내부 구조 및 구성 요소의 기능을 기술하고, 기존에 수행되었던 불량 블록 관리 기법에 대한 연구를 심화하여 설명한다. 제 3장에서는 불량 블록 관리 기법에 대한 가정을 수립하고 이를 바탕으로 발생 가능한 의존성 문제를 정리한다. 또한 의존성을 해결하기 위한 실행 규칙을 수립한다. 제 4장에서는 실행 규칙을 만족하면서 FTL요청을 병렬적으로 처리하는 설계 구조를 설명하고, 하드웨어로 구현 가능한 최적화 방안을 제시한다. 제 5장에서는 제안한 설계 방법에 따라

불량 블록 관리 기법을 구현하여 실험적으로 정확성을 검증하고 성능을 평가한다. 마지막으로 제 6장에서는 연구 결과를 요약하고 향후 연구 과제를 논의한다.

제 2 장　배경 지식 및 관련 연구

2.1 플래시 메모리

플래시 메모리는 전기적으로 데이터를 지우고 다시 기록할 수 있는 비휘발성 메모리의 일종으로서, DRAM과 같은 휘발성 메모리와 달리 전원이 공급되지 않아도 기록된 데이터가 유지되는 특성이 있다. 데이터를 저장하는 기본 단위인 셀의 연결 방식에 따라 NOR 플래시 메모리와 NAND 플래시 메모리로 구분된다. NOR 플래시 메모리는 바이트 또는 워드 단위의 임의 접근이 가능하지만 지우기 및 쓰기 속도가 느리기 때문에 컴퓨터 시스템의 부트로더 (Bootloader)나 내장형 시스템의 펌웨어 (Firmware)와 같이 자주 갱신되지 않는 실행 코드를 저장하는 용도로 사용된다. 이에 반해 NAND 플래시 메모리는 구조적인 특성으로 인해 임의 접근은 불가능하지만 셀의 집적도를 높이기 쉽기 때문에 단위 용량 당 가격이 저렴하며 전력 소모가 적다. 또한 쓰기 및 읽기의 단위가 크기 때문에 일정 규모 이상 (대개 수백 바이트에서 수 킬로바이트)의 순차적인 데이터 접근 시 NOR 플래시 메모리에 비해 월등히 높은 성능을 보인다. 이러한 장점으로 인해 NAND 플래시 메모리는 메모리 카드나 SSD (Solid State Drive)와 같은 대용량 저장 장치에서 널리 활용되고 있다. 현재 스마트폰과 같은 소형 모바일 시스템에서부터 대규모의 데이터를 관리하는 데이터 센터에 이르기까지 고성능 저전력 저장장치에 대한 수요가 폭발적으로 증가하고 있으며, 이에 따라 NAND 플래시 메모리의 중요성이 점점 커지고 있다. 본 논문에서는 NAND 플래시 메모리에 한정하여 논의를 진행하도록 한다.

2.2 플래시 메모리의 기본 원리

플래시 메모리의 기본 구성단위인 셀 (Cell)은 비휘발적으로 정보를 저장하기 위해 변형된 MOS (Metal Oxide Semiconductor) 트랜지스터의 일종이다. MOS 트랜지스터는 스위치 역할을 하는 반도체 소자이며, 기판 (Substrate) 위에 소스 (Source), 드레인 (Drain), 게이트 (Gate)라고 불리는 3개의 단자로 구성된다. 게이트에 정해진 임계 전압 (Threshold voltage) 이상의 전압이 가해지면 소스와 드레인 사이에 전하가 이동할 수 있는 채널이 형성되어 전류가 흐른다. 그림 1은 플래시 메모리 셀의 구조를 보여 준다. MOS 트랜지스터와 달리 두 개의 게이트를 가지고 있으며, 각각을 제어 게이트 (Control gate)와 부동 게이트 (Floating gate)라고 부른다. 제어 게이트는 MOS 트랜지스터의 게이트와 동일한 역할을 하며 부동 게이트는 내부에 전자를 저장함으로써 정보를 표현하는 역할을 한다. 부동 게이트 주변은 터널링 산화막 (Tunnel oxide)에 의해 절연되었기 때문에 외부에서 강한 전압을 가하지 않는 한 부동 게이트와 외부 사이의 전자 이동이 단절되며, 이로 인해 비휘발적으로 정보를 저장하는 것이 가능하다. 부동 게이트에 전자가 유입된 상태를 쓰여진 상태 (Programmed state)라고 하며 0이라는 비트 값이 저

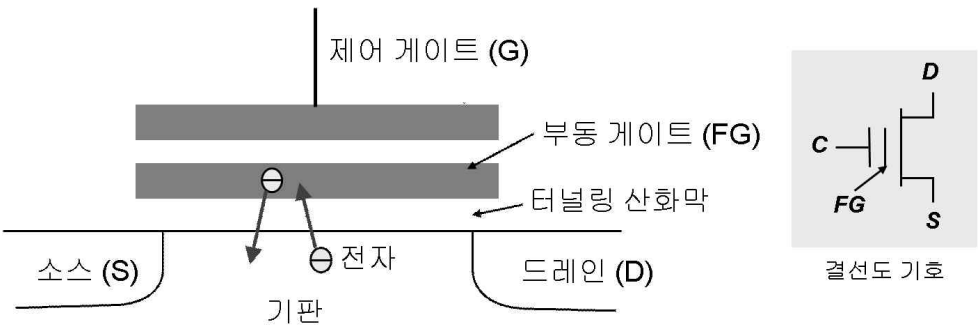


그림 1 플래시 메모리 셀의 구조[11]

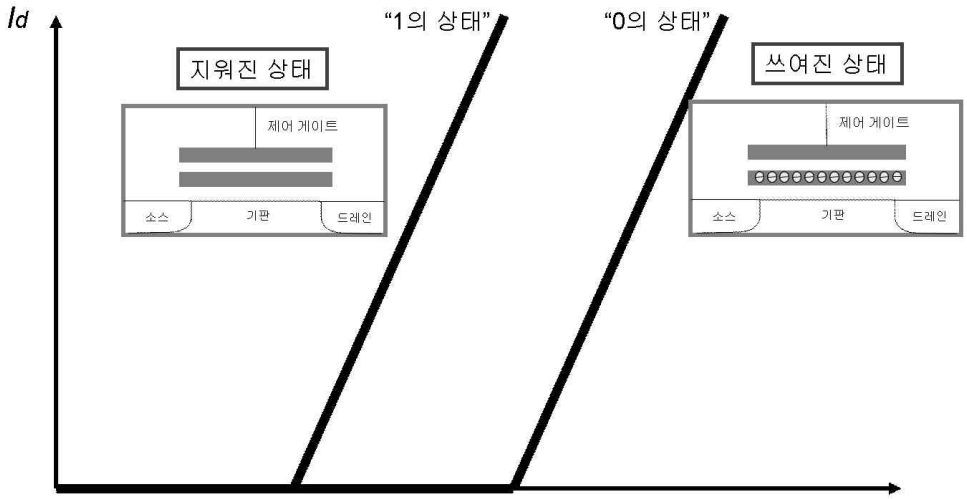
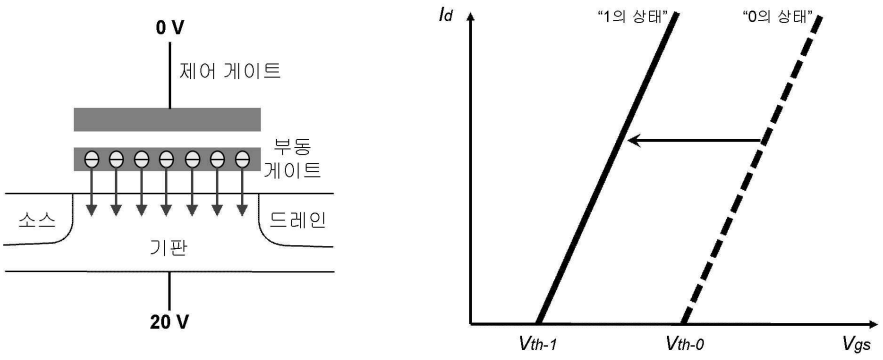


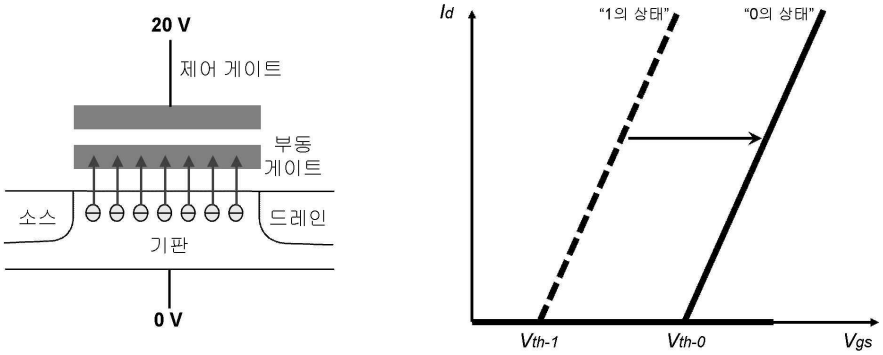
그림 2 셀의 상태 및 임계 전압

장되었음을 나타낸다. 반면 부동 게이트에서 전자가 방출된 상태를 지워진 상태 (Erased state)라고 하며 해당 비트의 값은 1을 의미한다. MOS 트랜지스터가 고정된 임계 전압을 가지는 것과 달리 플래시 메모리 셀은 부동 게이트 내의 전자의 양에 따라 임계 전압이 변동되며, 보유 전자의 양이 많을수록 높은 임계 전압을 가지게 된다. 그림 2는 플래시 메모리 셀의 상태에 따라 서로 다른 두 개의 임계 전압이 존재함을 보여준다.

플래시 메모리 셀의 상태를 전이시키기 위해서는 F-N 터널링 (Fowler-Nordheim tunneling) 기법을 사용한다[12]. F-N 터널링 기법은 제어 게이트와 기판 사이의 높은 전위차를 통해 부동 게이트와 기판 사이로 전자를 이동시켜서 셀의 임계 전압을 변경한다. 지우기 연산의 경우, 그림 3과 같이 기판에 상대적으로 높은 전압을 인가하여 부동 게이트의 전자를 방출시키며, 쓰기 연산의 경우 반대로 제어 게이트에 높



(a) 지우기 연산



(b) 쓰기 연산

그림 3 F-N 터널링에 의한 상태 전이[13]

은 전압을 인가하여 기판으로부터 부동게이트로 전자를 유입시킨다.

셀의 상태를 판별하여 저장된 비트 값을 읽기 위해서는 두 임계 전압 사이의 전압을 제어 게이트에 인가하는데 이와 같은 전압을 참조 전압 (Reference voltage) 라고 한다. 셀의 임계 전압이 Vth-1이면 참조 전압이 임계 전압보다 높으므로 드레인과 소스 사이에 전류가 흐르게 되고, Vth-0이면 전류가 흐르지 않는다. 이와 같은 전류의 흐름 여부를 감별하여 1의 상태 (지워진 상태)와 0의 상태 (쓰여진 상태)를 구별하게 된다.

NAND 플래시 메모리에서는 개별 셀에 대하여 독립적으로 연산을 수행하는 것이 불가능하며, 페이지 (Page) 나 블록 (Block) 이라고 불리는 셀들의 그룹에 대해서만 연산을 수행할 수 있다. 페이지는 쓰기나 읽기 연산의 단위이며, 페이지의 크기는 플래시 메모리 제품별로 상이하나 보통 2/4/8 킬로바이트의 데이터 저장 영역과 수십~수백 바이트의 여분의 영역을 제공한다. 블록은 지우기 연산의 단위로서 보통 64/128/256 개의 페이지로 구성되며, 단일 플래시 메모리 칩에는 수천 개의 블록들이 존재한다. 그림 4는 64개의 페이지로 구성된 플래시 메모리의 블록 구조를 보여 준다. 블록 내에는 플래시 메모리 셀들이 2차원 행렬을 이루고 있으며 동일한 행에 속하는 셀들의 집합이 읽기/쓰기의 단위인 페이지이다. 페이지 내의 셀들은 제어 게이트에 전압을 인가하기 위한 워드 라인 (Word line)을 공유한다. 한편 동일한 열을 구성하는 셀들의 집합을 스트링 (String) 이라고 하며, 스트링 내의 인접한 셀은 소스와 드레인을 공유하기 위하여 직렬로 연결되어 있다. 이와 같이 소스/드레인이 공유되는 구조와 페이지 내 워드 라인이 공유되는 구조로 인해 NOR

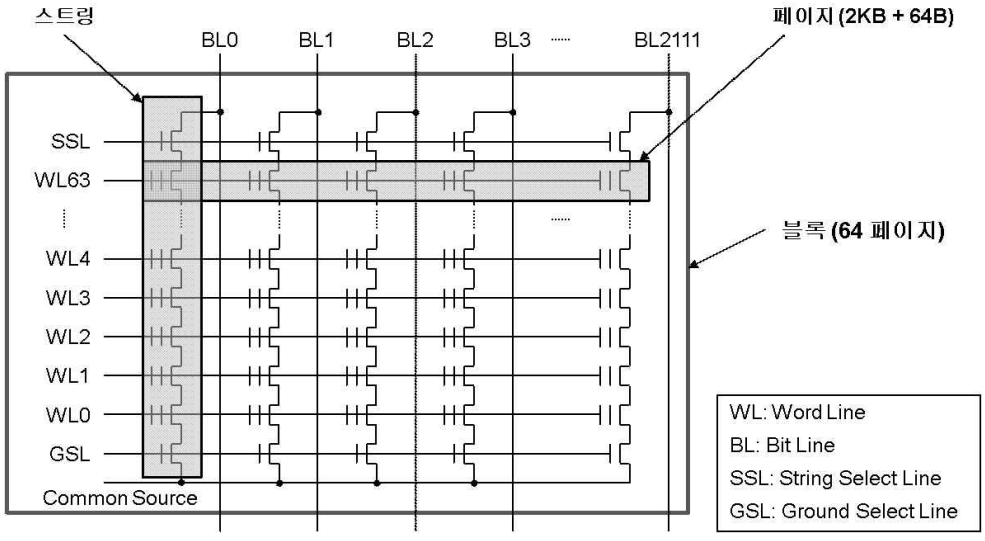


그림 4 플래시 메모리 블록의 구조

플래시 메모리에 비해 단위 면적 당 많은 수의 셀을 집적시킬 수 있으며 대용량의 NAND 플래시 메모리 칩이 가능하게 된다. 그러나 개별 셀에 대한 독립적인 접근이 불가능하고, 지우기 단위와 쓰기 및 읽기 단위의 비대칭성으로 인해 플래시 메모리를 효율적으로 사용하기 위해서는 FTL (Flash Translation Layer)과 같은 별도의 소프트웨어 관리 계층이 필요하다.

플래시 메모리 블록을 선택하여 지우기 연산을 수행하는 경우, 블록의 기판에 20V를 걸고 모든 워드 라인에 0V를 걸어 준다. 이 때, 지우기 연산이 일어나지 않는 다른 블록들은 기판과 워드 라인에 20V를 동일하게 인가한다.

쓰기 연산은 지우기에 비해 복잡한데, 블록 내에 쓰기 연산이 수행되는 페이지와 그렇지 않은 페이지를 구분해야 하고, 페이지 내에서도 기록하려는 값이 1인 셀인 쓰기 금지시켜야하기 때문이다. 워드 라인에 인가되는 전압을 살펴보면, 쓰기 대상 페이지의 경우 20V이고 다른 페이지들은 10V이다. 기판에 대한 전압은 비트 라인을 통해 인가하는데 이는 페이지 내의 셀 별로 다른 전압을 가해야하기 때문이다. 대상 페이지 내의 0이 쓰여 지는 셀과 연결된 비트 라인에는 0V를 가하고 그 외의 비트 라인은 쓰기 금지를 위해 10V를 가한다. 대상 페이지 내 쓰기 금지될 셀들은 제어 게이트와 기판 사이에 10V 전위차가 발생하지만 F-N 터널링을 일으키기에 충분한 전위차가 아니므로 쓰기 연산이 일어나지 않는다. 쓰기 대상 페이지가 아닌 셀들은 동일 스트링 내에 쓰기 연산이 일어나는 셀의 존재 여부에 따라 제어 게이트와 기판 사이의 전위차가 0V 또는 10가 되며, 두 경우 모두 F-N 터널링을 발생시키기에 부족한 전위차이므로 쓰기 연산은 일어나지 않는다.

플래시 메모리의 제조 공정이 20nm 대로 미세화 됨에 따라 셀에 저장된 데이터의 신뢰성을 보장하기 위해 쓰기 연산 시 부동 게이트에 주입하는 전자의 수를 보다 정밀하게 제어할 필요성이 커지고 있다. 이와 같은 목적을 달성하기 위해 현재 대부분의 플래시 메모리 칩에서는 ISPP (Incremental Step Pulse Programming) 라는 쓰기 알고리즘을 사용하고 있다[14][15]. ISPP 알고리즘에서는 부동 게이트에 주입된 전자의 양이 과도하거나 부족한 상황을 방지하기 위하여, 적은 양의 전자를 주입하는 작업과 주입된 전자의 양을 확인하는 과정을 반복하는 방식을 취한다. 전자 주입을 위해 단계적으로 크기가 증가하는 전압 펄스(Pulse) 를 제어 게이트에 인가하며, 전압 펄스 사이에 주입된 전자의 양을 확인하기 위한 참조 전압을 인가하여 셀의 임계 전압이 목표한 수준에 도달했는지 확인하다. 임계 전압이 참조 전압보다 높아지면 해당 셀에 대한 쓰기 작업을 종료한다.

페이지 읽기를 수행하는 경우, 대상 페이지의 워드 라인에는 Vth-1과 Vth-0 사이의 참조 전압을 가하여 쓰여진 상태에 있는 셀들만 드레인과 소스 사이에 전류가 흐르도록 한다. 그 외 페이지에 대해서는 Vth-0보다 높은 전압을 가하여 셀의 상태와 상관없이 전류가 흐르도록 한다. 결과적으로 읽기 대상 페이지 중 쓰여진 상태의 셀이 포함된 스트링에서만 전류가 흐르게 되고, 전류 흐름 여부를 감지하여 페이지 내의 데이터 값을 읽게 된다.

지금까지 하나의 셀에 한 비트의 데이터를 저장하는 경우에 대해서 살펴보았다. 부동 게이트 내의 전자의 양을 보다 세밀하게 조절한다면 더 많은 수의 상태를 표현할 수 있으며 하나의 비트가 아닌 여러 비트를 셀에 저장하는 것이 가능해진다. 하나의 셀이 하나의 비트를 표현하는

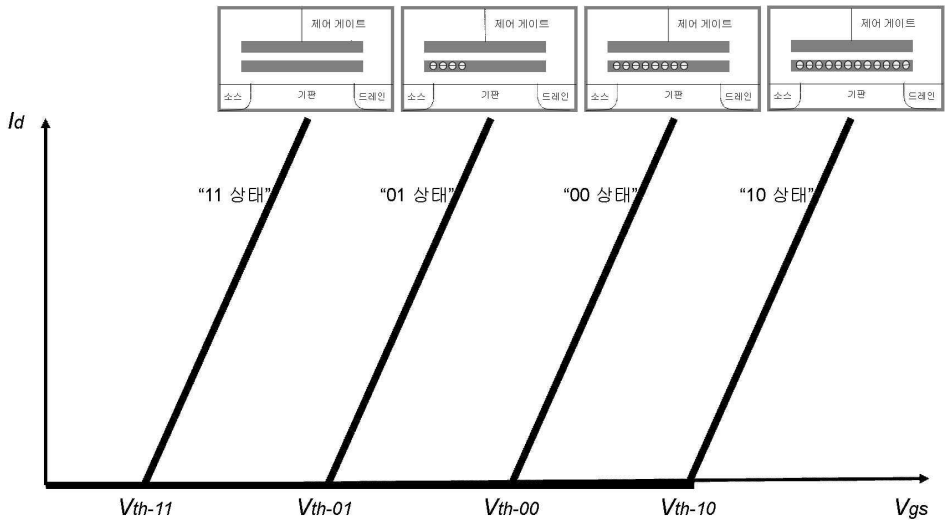


그림 5 MLC 셀의 상태 (셀 당 2비트를 저장)

플래시 메모리를 SLC (Slingle-Level Cell) 플래시 메모리라고 하며, 2개 이상의 비트를 표현하는 것을 MLC (Multi-Level Cell) 플래시 메모리라고 한다. 그림 5는 2 비트를 표현하는 MLC 플래시 메모리에서의 셀 상태를 보여준다. 부동 게이트에 유입된 전자의 양에 따라 네 가지 상태를 가지며 각각 11, 01, 00, 10 상태라고 명명한다. 또한 상태 개수만큼의 임계전압을 가지게 되며, 읽기 작업을 위해 게이트에 인가하는 참조 전압의 수도 세 가지로 늘어난다. SLC와 달리 참조 전압 개수가 복수 개이기 때문에 여러 번의 참조 전압을 인가해야 저장된 비트 값들을 읽을 수 있다. 일반적으로 K 비트 MLC의 경우, 플래시 메모리 셀의 상태는 2^K 개, 참조 전압은 2^{K-1}개로 급격히 증가한다.

그림 6은 플래시 메모리 셀이 표현하는 비트 수가 증가함에 따른 플래시 메모리 셀의 임계 전압 분포를 나타낸다.[1] 가능한 임계 전압의 범

1) 동일한 비트 값을 표현하는 상태라고 하더라도 임계 전압은 셀 별로

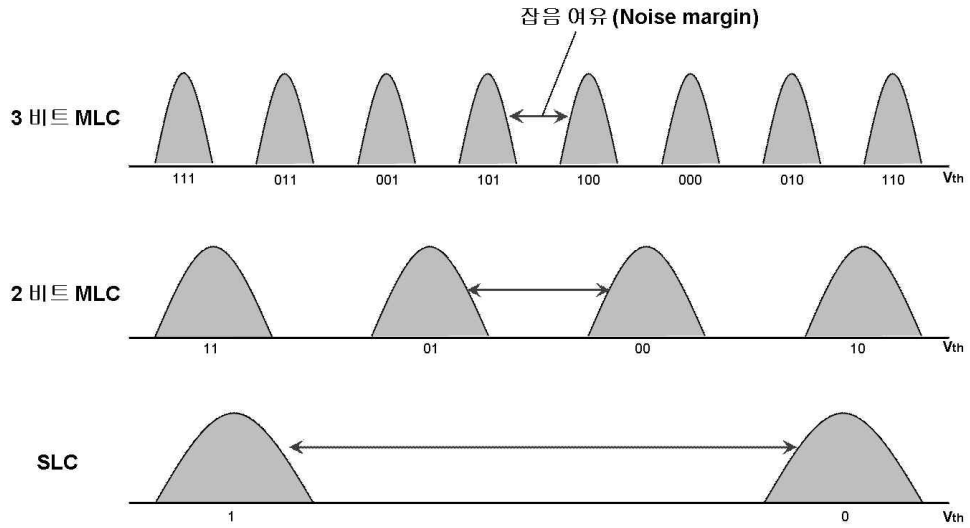


그림 6 셀 당 비트 수에 따른 잡음 여유

위는 제한되어 있으므로, 셀의 상태 수가 증가하게 되면 서로 다른 임계 전압 간의 간격이 좁아지게 된다. 이는 플래시 메모리에 데이터를 쓰는 경우 부동 게이트에 유입되는 전자의 양을 보다 정밀하게 제어해야 하고, 읽기 시에는 제어 게이트에 인가되는 참조 전압의 크기를 정밀하게 제어할 뿐만 아니라 드레인과 소스 사이에 흐르는 전류도 보다 정교하게 감지해야 함을 의미한다. 따라서 MLC 플래시 메모리는 SLC 플래시 메모리에 비해 읽기 및 쓰기 알고리즘이 복잡하며 이로 인해 플래시 메모리 연산에 대한 지연 시간이 길어진다. 또한 다양한 원인에 의해 일부 플래시 메모리 셀의 임계 전압이 이동 (Voltage shift)하는 현상이 발생

상이할 수 있다. 부동 게이트에 유입된 전자의 양이나 터널링 산화막의 두께와 같은 임계 전압을 결정하는 요인이 셀 별로 다르기 때문이다. 따라서 전체 셀의 실제 임계 전압은 그림 6과 같은 분포의 양상을 가지게 된다.

하는 경우, MLC 플래시 메모리는 임계 전압 간의 간격이 좁기 때문에 셀의 상태가 변경될 확률이 높아진다. 이는 MLC 플래시 메모리가 SLC에 비해 저장된 데이터의 신뢰성 측면에서 취약함을 나타내며 보다 많은 비트 오류를 정정하는 ECC (Error Correction Code)회로가 필요함을 의미한다. 그러나 MLC 플래시 메모리는 SLC 플래시 메모리에 비해 단위 면적당 저장 용량을 획기적으로 늘릴 수 있는 장점이 있기 때문에, 가격 경쟁력이 중요한 모바일 제품이나 개인용 컴퓨터를 기반으로 그 응용 범위가 점점 확대되고 있다.

2.3 플래시 메모리의 신뢰성 특성

플래시 메모리의 반도체 제조 공정이 20nm 급으로 세밀해지고 MLC 기술이 보편적으로 사용됨에 따라 플래시 메모리의 신뢰성이 심각한 문제로 부각되고 있다. 플래시 메모리는 다양한 요인에 의해 데이터 오류를 발생시킬 수 있는데, 셀 자체의 결함에 의한 오류와 일시적인 오동작에 의한 오류로 구분할 수 있으며, 오류의 유형에 따라 다른 종류의 오류 복구 기술을 적용해야 한다.

플래시 메모리 셀은 제조 공정 중 미세 먼지나 금속 불순물과 같은 오염원에 의해 손상될 수 있다. 또한 정상적인 셀이라도 지우기와 쓰기 연산이 반복적으로 수행되게 되면 부동 게이트와 기판 사이의 터널 산화막에 결함이 생겨 데이터를 저장하거나 읽을 수 없게 된다. 그림 7은 정상 셀의 마모 과정을 보여 준다. 지우기와 쓰기 연산이 되풀이됨에 따라 기판이나 부동 게이트로 완전히 이동하지 못하고 터널링 산화막에 잔류하는 전자가 생기게 되는데, 이와 같은 현상을 전하 포획 현상 (Charge

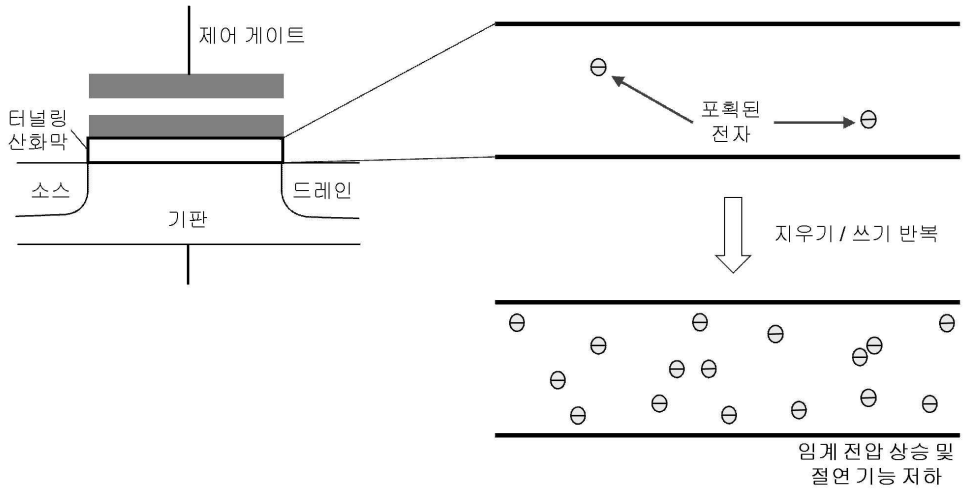


그림 7 지우기/쓰기 반복에 의한 셀의 마모 과정

trapping mechanism) 이라고 한다[16][17]. 터널링 산화막 내에 포획된 전자는 부동 게이트 내의 전자와 마찬가지로 임계 전압을 상승시키게 되며, 포획된 전자의 양이 일정 수준을 초과하게 되면 지우기나 쓰기 연산 시 임계 전압이 충분히 내려가지 못하거나 과도하게 올라가는 오류가 발생하게 된다. 또한 포획된 전자에 의해 터널 산화막의 절연 특성이 약화되는데 이에 따라 지우기와 쓰기 연산을 수행하지 않는 경우에도 부동 게이트와 기판 사이에 전자가 이동할 가능성이 커지게 된다. 이는 플래시 메모리 셀의 비트 오류율이 증가함을 의미한다. 플래시 메모리 제조 공정이 미세화되면 부동 게이트에서 유지하는 전자의 수가 줄어들기 때문에 포획된 전자에 의한 영향이 상대적으로 커지게 된다. 마찬가지로 MLC의 경우, 임계 전압 간의 간격이 좁기 때문에 포획된 전자에 의한 임계 전압 이동에 취약해진다. 따라서 플래시 메모리의 집적도가 증가하게 되면 지우기나 쓰기 반복에 대한 셀의 내구성 (Endurance) 이 감소하게 된다. 이러한 내구성은 플래시 메모리 제품의 수명을 결정하는 중

요한 요인이기 때문에 대부분의 제조사에서는 내구성의 척도로서 허용 가능한 지우기/쓰기 횟수를 명시하고 있다. 통상 SLC 플래시 메모리의 경우 100,000 번, 2 비트 MLC 플래시 메모리는 3,000 ~ 10,000 번, 3 비트 MLC 플래시 메모리는 1,000 ~ 5,000 번의 지우기와 쓰기 반복을 허용하나, 플래시 메모리의 집적도가 증가함에 따라 이러한 내구성 수치도 점점 감소하는 추세이다

플래시 메모리 제조 공정 중에 손상되었거나 내구성 한계를 초과하여 더 이상 사용할 수 없는 셀을 일정 개수 이상 포함하는 블록을 불량 블록이라고 하며, 발생 원인에 따라 초기 불량 블록 (Initial bad block, 제조 공정 중에 셀 손상)과 동작 중 불량 블록 (Runtime bad block, 지우기 및 쓰기 연산의 반복으로 인한 셀 손상)으로 구분한다. 불량 블록은 연산에 대한 정확한 동작을 보장하지 못하기 때문에 저장 장치 시스템의 신뢰성을 위해 불량 블록을 식별하고 시스템 상에서 배제하는 작업이 필수적이며, 이와 같은 기능을 수행하는 기법을 불량 블록 관리 기법 (Bad block management scheme) 이라고 한다[9][10]. 또한 지우기, 쓰기가 일부 블록 영역에 집중되어 특정 블록들이 급속하게 마모되면 시스템 전체적으로 가용한 정상 블록들이 조기에 소진되게 된다. 이러한 현상을 방지하기 위해 모든 블록에 대해 균등하게 지우기와 쓰기 연산을 수행하는 작업이 필요하며 이를 마모도 평준화 (Wear-leveling) 기법이라고 한다[5][18][19].

정상적인 블록이라고 하더라도 일시적인 오동작에 의해 저장된 데이터의 비트 오류가 발생할 수 있다. 대표적인 예로 초과 쓰기 (Over-programming) 오류가 있다[20]. 플래시 메모리에서는 쓰기 연산 시 부동 게이트에 주입되는 전자의 수를 정밀하게 제어하기 위해 쓰기와

확인 과정을 반복하는 ISPP 알고리즘을 사용하는데, 터널링 산화막의 절연성이 약화된 상태의 셀은 쓰기 과정 중에 과도한 양의 전자가 주입되어 임계 전압이 의도한 수준을 초과할 수 있다[22]. 또한 확인 과정 중에 무작위 전송화 잡음 (Random telegraph noise)이 발생하여 드레인과 소스 사이의 전류를 제대로 감지하지 못하는 경우, 불필요한 쓰기 과정을 수행하게 되어 초과 쓰기가 발생하기도 한다[21]. 이와 같은 초과 쓰기는 쓰기 연산 시 부동게이트로 전자를 주입하는 것은 가능하나 방출시키는 것은 불가능한 플래시 메모리 셀의 구조적 한계에 기인한다. 초과 쓰기에 의해 대상 페이지에는 일부 데이터 비트의 값이 반전되는 비트 오류가 발생할 수 있으며, 이러한 오류를 정정하기 위해 플래시 메모리 저장 장치 시스템에서는 ECC (Error Correction Code) 회로가 필수적으로 사용된다[23][24][25][26][27]. 초과 쓰기로 인한 데이터 오류는 일시적인 것으로써 지우기 연산을 통해 그 효과가 제거되며, 셀의 영구적인 손상에 기인한 것이 아니기 때문에 추후 정상적으로 사용될 수 있다.

정상적으로 쓰여진 페이지의 경우에도, 다양한 요인에 의해 비트 오류가 발생할 수 있다. 쓰기 과정 중에 동일 블록 내에서 이전에 쓰여진 페이지에 속하는 셀들은 기판과 제어 게이트 사이에 작은 크기의 전위차 발생할 수 있는데, 터널링 산화막의 절연성의 한계로 인해 작은 크기의 전위차에도 소량의 전자가 부동게이트에 주입될 수 있다. 이러한 현상을 쓰기 교란 (Program disturbance) [28]라고 하며, 쓰기 대상 페이지 내의 쓰기 금지될 셀에서도 동일한 원리에 의해 쓰기 교란이 발생할 수 있다. 쓰기 교란은 초과 쓰기와 마찬가지로 임계 전압이 의도한 것보다 상승하는 결과를 초래하며, 쓰기 교란이 누적되면 비트 오류가 발생할 수

있다. 초창기 SLC 플래시 메모리에서는 한 페이지에 복수 번의 부분 쓰기 (Partial programming) 를 허용하였으나, 부분 쓰기로 인해 쓰기 횟수가 증가하게 되면 쓰기 교란 발생 가능성이 커지므로, 비트 오류에 취약한 현재 대부분의 플래시 메모리에서는 부분 쓰기를 허용하지 않는다[29].

교란 현상은 읽기 연산 중 에도 발생하는데, 읽기 연산 시 동일 블록 내의 다른 페이지에 속하는 셀에 통과 전압이 인가되어 부동 게이트에 소량의 전자가 주입될 수 있으며 이를 읽기 교란 (Read disturbance)[28]이라고 한다. 읽기 교란도 쓰기 교란과 마찬가지로 임계 전압 상승에 의한 비트 오류를 유발할 수 있다.

교란 현상 이외에도 인접한 셀 간의 간섭에 의해 임계 전압이 의도치 않게 상승할 수 있다. 인접한 셀의 부동 게이트 사이에는 기생 용량 (Parasitic capacitance)이 결합 (Coupling)되어 존재하며, 특정 셀에 쓰기 연산을 수행하게 되면 기생 용량 결합으로 인해 주변 셀들의 부동 게이트에도 전자가 주입되는 현상이 발생한다[30]. 이와 같은 간섭 현상으로 인한 오류는 플래시 메모리의 집적도가 증가함에 따라 점점 심각한 문제로 부각되고 있으며, 이와 같은 문제를 완화하기 위해서 플래시 메모리에 쓰기 연산 수행 시 쓰기 순서에 대한 제약을 도입하였다. 동일 블록 내 페이지들은 항상 페이지 번호가 증가하는 순서로만 쓰기 작업이 수행되어야 하고[31], 이와 같은 오름차순 쓰기 (Ascending-order programming) 제약을 지키지 않으면 데이터의 비트 오류가 과도하게 증가하여 ECC로 복구가 안 되는 데이터 손실 문제가 발생할 수 있다.

데이터 오류는 플래시 메모리 상에서 연산이 진행되지 않는 유휴 상

태 (Idle state)에서도 발생할 수 있다. 플래시 메모리 셀에 전압을 가하지 않는 상황에서도 터널링 산화막의 결함에 의해 부동 게이트에 축적된 전자가 기판으로 빠져나가는 누설 전류가 발생할 수 있는데, 이를 SILC (Stress Induced Leakage Current)라고 한다. SILC는 임계 전압이 높은 상태의 셀에서 주로 발생하며, SILC로 인한 셀의 임계 전압 하강이 누적되면 비트 오류가 유발된다[32][33]. 터널링 산화막에 포획되어 있는 전자도 시간에 따라 기판으로 방출될 수 있는데 이를 전하 탈출 (Charge detrapping)이라고 하며 SILC와 마찬가지로 셀의 임계 전압이 내려가는 원인이다[17][34]. 이와 같이 유휴 상태에서 부동 게이트나 터널링 산화막의 전자 소실로 인해 발생하는 오류를 데이터 유지 오류 (Data retention error)라고 한다.

[20]은 2 비트 MLC 플래시 메모리의 비트 오류율 (Bit Error Rate; BER)을 실험적으로 분석하였다. 지우기와 쓰기의 반복으로 인해 페이지의 마모가 진행되면 초과 쓰기, 교란, 데이터 유지 오류에 의한 비트 오류 발생 빈도가 증가함을 확인하였다. 그러나 이러한 오류는 대부분 일시적인 것으로써, 비트 오류가 발생한 셀이라고 하더라도 다음 번 지우기와 쓰기 연산 수행 시 반복적인 비트 오류는 발생하지 않았다. 또한 읽기 교란에 의한 비트 오류는 다른 오류 메커니즘에 비해 발생 빈도가 적었으며, ECC 사용을 가정하였을 때 단일 오류 메커니즘에 의해서 ECC로 정정 불가능한 비트 오류가 발생할 확률을 극히 적었다. 그러나 쓰기 교란이 발생한 페이지에 데이터 유지 오류가 누적되는 것과 같이 서로 다른 메커니즘이 복합적으로 발생하면 ECC를 사용하더라도 비트 오류로 인해 데이터가 손실될 위험이 급격히 증가하였다.

[35]는 지우기와 쓰기 횟수에 따른 SLC와 2 비트 MLC 플래시 메

모리의 비트 오류율을 실험하였다. SLC 플래시 메모리는 데이터시트 상에서 명시하고 있는 내구성 한계 (지우기/쓰기 횟수 100,000 번)에 도달할 때까지 비트 오류율이 0에 가까운 수준을 유지하였다. 또한 내구성 한계를 훨씬 초과한 지우기/쓰기 600,000 번에 도달할 때까지 낮은 수준의 비트 오류율을 보였다. 반면, MLC 플래시 메모리의 경우, 마모가 거의 없는 상태에서도 이미 높은 비트 오류율을 보였으며 내구성 한계인 지우기/쓰기 10,000 번에 도달하기도 전에 비트 오류율이 급격히 증가하는 현상을 보이기도 하였다.

[35]는 2 비트 MLC 플래시 메모리에서 블록 내 페이지 중 절반이 나머지에 비해 비트 오류율이 훨씬 낮은 현상을 관찰하였다. 이는 서로 다른 페이지가 셀을 공유하는 MLC 플래시 메모리의 물리적인 구조에 기인한다. 2 비트 MLC 플래시 메모리의 예를 들면, 하나의 셀을 통해 표현되는 2개의 비트 중 상위 비트 (MSB; Most significant bit)와 하위 비트 (LSB; Least significant bit) 는 서로 다른 두 개의 페이지에 속하게 되며, 하위 비트로 구성되는 페이지는 상위 비트 페이지에 비해 성능이나 신뢰성 측면에서 우수하다. 이와 같이 서로 다른 페이지가 셀을 공유하는 구조와 이로 인한 성능/신뢰성 측면의 편차는 모든 MLC 플래시 메모리의 공통적인 특성이다. 셀을 공유하는 페이지들을 짝 페이지 (Paired page) 또는 형제 페이지 (Sibling page)라고 하며, 형제 페이지들 간에는 쓰기 연산 실패로 인한 치명적인 데이터 손상 (Data corruption)이 발생할 수 있다. 형제 페이지 관계를 이루는 두 페이지 중, 하위 비트 페이지가 정상적으로 쓰여진 후, 상위 비트 페이지에 대한 쓰기 연산을 진행하는 도중 전원 오류나 쓰기 오류가 발생하게 되면 정상적으로 기록된 하위 비트 페이지의 데이터가 손상되게 된다. [36]에서는 이와

같은 문제를 소급 데이터 손상 (Retroactive data corruption) 문제라고 명명하였으며, 실험을 통해 이러한 문제가 발생함을 확인하였다. 소급 데이터 손상 문제는 다른 비트 오류 문제와는 달리 ECC를 통해 해결할 수 없으며, FTL과 같은 소프트웨어에서 데이터 손상을 사전에 방지하거나 손상된 데이터를 복구하기 위한 추가적인 기법을 사용해야 한다.

2.4 병렬 연산 처리를 위한 플래시 메모리의 내부 구조

플래시 메모리의 집적도가 증가함에 따라 단일 플래시 메모리 연산의 실행 속도는 점점 감소하고 있다. 초미세 반도체 제조 공정과 MLC 기술이 보편적으로 적용되면서 플래시 메모리 셀의 상태를 제어하는 알고리즘이 복잡해지고 있기 때문이다. 이는 플래시 메모리가 전체 컴퓨터 시스템 성능에 심각한 병목 (Bottleneck) 현상을 초래할 수 있음을 암시한다. 더욱이 대규모 데이터 센터와 같이 단위 시간 당 막대한 양의 저장 장치 입출력을 요구하는 응용에서는 플래시 메모리의 연산 속도 저하가 중대한 도전 과제가 되고 있다. 이와 같은 문제를 해결하기 위해 복수의 연산을 병렬적으로 실행함으로써 연산의 처리율 (Throughput) 을 높이기 위한 플래시 메모리 설계 기술이 사용되고 있다.

그림 8은 한 번에 하나의 연산을 실행하는 초창기 플래시 메모리의 구조를 보여준다. 데이터를 비휘발적으로 저장하기 위한 블록 및 페이지 구조의 플래시 메모리 셀 영역 뿐만 아니라 연산을 수행하기 위한 다양한 레지스터들이 있다. 명령 레지스터 (Command register)는 수행하려는 연산의 종류를 명시하기 위한 것으로써 연산을 시작하려면 명령 레지스터로 명령 코드 (지우기/쓰기/읽기 등)를 전달해야 한다. 주소 레지스

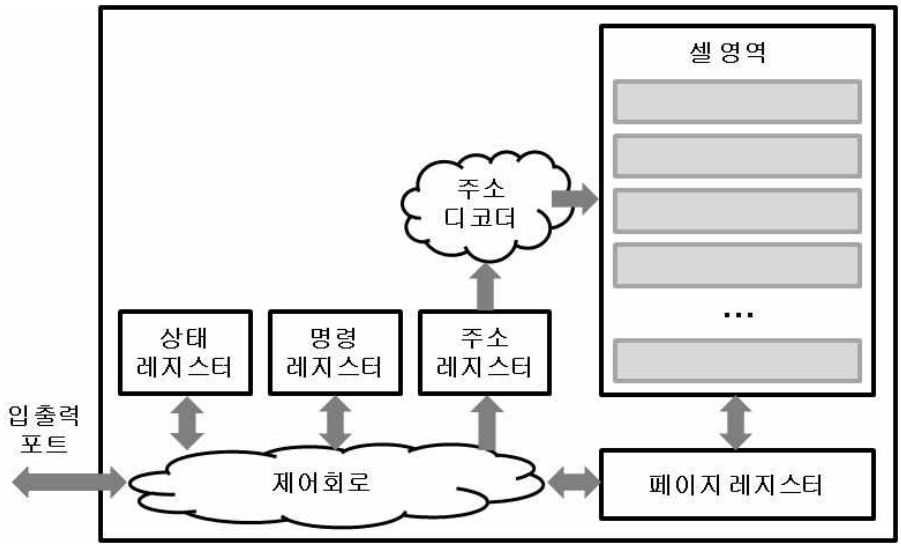


그림 8 초창기 플래시 메모리의 내부 구조

터 (Address register)는 접근하려는 블록이나 페이지의 주소를 지정하는 역할을 한다. 페이지 레지스터 (Page register) 는 쓰기나 읽기 연산 시 페이지 단위의 데이터를 임시로 저장 (Buffering)하는 용도로 사용된다. 이는 플래시 메모리 칩의 입출력 포트 (I/O port) 로는 바이트 단위의 데이터 전송이 일어나지만 플래시 메모리 셀 영역으로는 페이지 단위의 데이터 읽기/쓰기만이 허용되는 접근 단위의 불일치를 해결하기 위함이다. 쓰기 연산 시 사용자의 데이터는 입출력 포트를 통해 페이지 레지스터로 전송되고, 입출력이 완료되면 페이지 레지스터에서 셀 영역으로 쓰기 작업이 일어난다. 읽기 연산은 이와 정반대의 순서로 진행되는데, 셀 영역에서 페이지 레지스터로 데이터가 적재 (Load)된 후, 입출력 포트를 통해 페이지 레지스터에서 외부로 데이터가 전달된다. 그 밖에 플래시 메모리 셀 영역에 대한 연산의 진행 상태나 지우기 및 쓰기 시 발생한 오류 여부를 알려 주는 상태 레지스터 (Status register)가 존재한다.

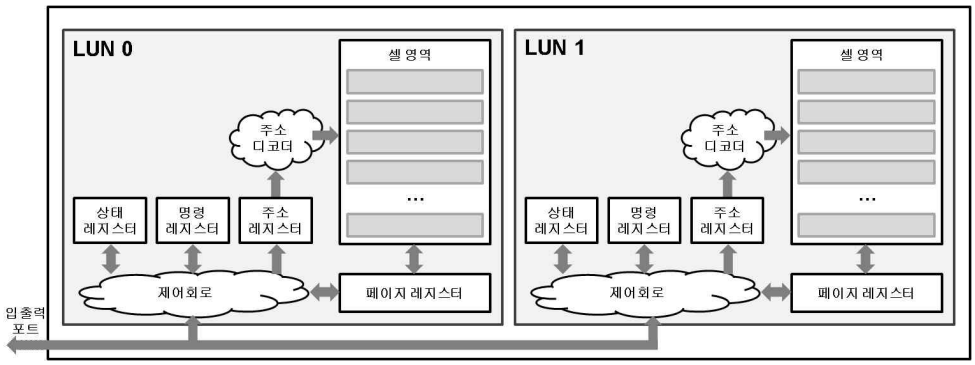


그림 9 다중 LUN 구조

최신 플래시 메모리에서는 여러 개의 독립적인 연산을 병렬적으로 수행하기 위해 전체 칩을 여러 개의 논리적인 칩으로 분할하는데, 이를 LUN (Logical Unit Number) 또는 die 라고 한다[37]. 개별 LUN은 연산 수행을 위해 필요한 레지스터 집합 (명령/주소/상태 레지스터 및 페이지 레지스터)과 고유의 셀 영역 및 제어 회로를 가지고 있기 때문에, 여러 개의 LUN이 독립적으로 연산을 수행하는 것이 가능하다. 그러나 외부로의 입출력 포트를 LUN들이 공유하고 있기 때문에 입출력 포트를 통해 명령/주소/데이터를 전송하는 작업은 한 번에 하나의 LUN에 대해서만 이루어진다. 입출력 포트와 LUN의 연결 부분에는 입출력의 대상 LUN을 선택하기 위한 회로가 있으며, 블록 주소 비트 중 지정된 위치의 비트의 값에 따라 LUN을 선택한다. (대부분 블록 주소 내의 최상위 비트가 LUN을 지정하는 비트이다) 기존의 지우기/쓰기/읽기 연산들은 대상 블록의 주소를 플래시 메모리 칩에 전달하는 과정에서 자연스럽게 대상 LUN이 선택되지만 연산의 진행 결과 및 오류 여부를 확인하기 위한 상태 읽기 (Read status) 연산은 블록 주소를 동반하지 않기 때문에 대상 LUN을 선택할 수 없는 문제가 있다. 이를 위해 복수의 LUN을 지원

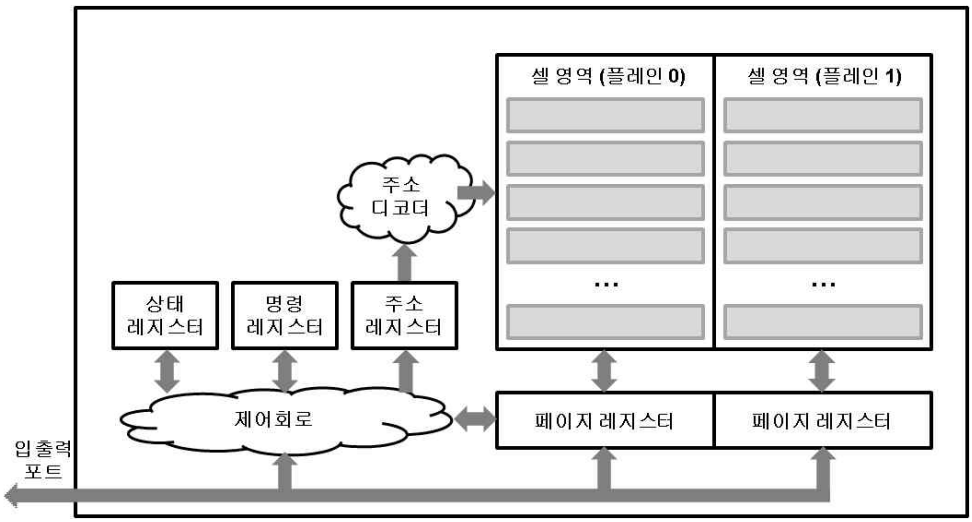


그림 10 다중 플레인 구조

하는 칩에서는 LUN 지정이 가능하도록 보완된 상태 읽기 (Read status enhanced) 연산을 지원한다.

LUN내의 블록들은 복수 개의 플레인 (Plane)으로 분할될 수 있으며, 서로 다른 플레인의 속한 블록들은 동시에 연산이 수행될 수 있다. 이를 위해 각각의 플레인은 독자적인 페이지 레지스터를 가지고 있다[37]. 그러나 동일 LUN내의 플레인 간에는 명령/주소/상태 레지스터를 공유하기 때문에, 플레인들을 완전히 독립적으로 구동시키는 것은 불가능하다. 서로 다른 플레인들을 동시에 구동시키기 위해서는 각 플레인의 동일한 블록 및 페이지 주소에 대해서 항상 동일한 종류의 연산을 수행해야 한다. 또한 LUN과 마찬가지로 입출력 포트를 공유하기 때문에 한 순간에 하나의 플레인에 대한 입출력이 가능하다. 플레인을 병렬적으로 구동하려면 별도의 다중 플레인 명령 코드를 사용해야 하는데, 이는 다중 플레인을 사용하지 않는 기존의 연산과 명확히 구분되기 위함이다. 다중 플레

인 연산은 연산의 종류와 접근 위치에 대한 제약으로 인해 LUN에 비해 병렬 처리에 대한 활용도가 떨어진다. 또한 별도의 명령 코드를 사용해야 하므로 FTL이나 플래시 메모리 제어기에서 이를 지원하기 위한 추가의 구현 복잡도를 감수해야 한다. 그러나 페이지 레지스터 이외의 모든 레지스터 및 제어 회로를 공유하기 때문에 LUN에 비해 플래시 메모리 칩의 구현 비용을 줄일 수 있고, FTL이나 파일시스템에서 대량 순차 접근 (Large sequential access) 위주의 작업 부하 (Workload)를 생성하는 경우 다중 플레인 연산을 효과적으로 활용할 수 있다.

연속적인 쓰기나 읽기 연산을 파이프라이닝 (Pipelining) 기법을 통해 중첩 실행하는 방법도 최신 플래시 메모리에서는 널리 사용되고 있다. 쓰기와 읽기 연산은 페이지 레지스터로 데이터 입출력을 수행하는 작업과 페이지 레지스터와 셀 영역 사이에 데이터를 전달하는 작업으로 이루어진다. 추가의 페이지 레지스터를 도입하면 이와 같은 두 작업을 중첩적으로 실행할 수 있는데, 이를 캐시 연산 (Cache operation)이라고 하며 추가로 사용되는 페이지 레지스터를 캐시 레지스터 (Cache register)라고 한다[37]. 캐시 쓰기는 세 단계의 작업을 거쳐 수행되는데, 입출력 포트를 통해 데이터를 캐시 레지스터로 전송하는 단계, 캐시 레지스터의 데이터를 페이지 레지스터로 복사하는 단계, 페이지 레지스터의 데이터를 셀 영역에 기록하는 단계로 이루어진다. 첫 번째 단계와 세 번째 단계는 페이지 레지스터에 대한 자원 충돌 (Resource conflict)이 없기 때문에 중첩되어 실행될 수 있다. 즉, K 번째 쓰기 연산의 셀 영역에 대한 기록 작업이 일어나는 동안에 K+1 번째 쓰기 연산을 위한 캐시 레지스터의 입출력 작업이 동시에 실행되는 것이다. 캐시 쓰기는 기존 쓰기에 비해 레지스터 간의 데이터 복사 작업이 부가되었지만, 복사로 인한 지

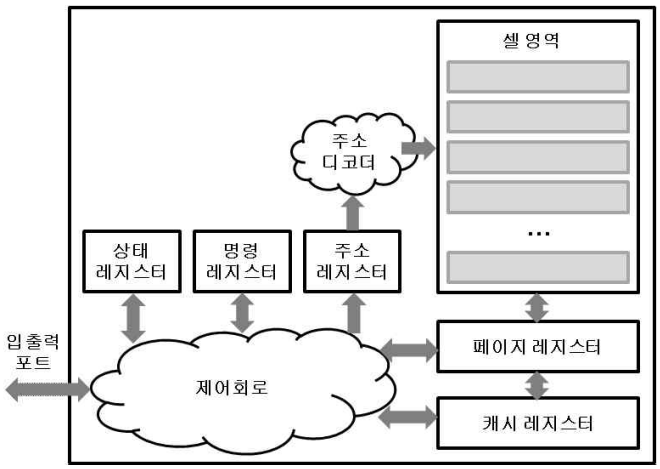


그림 11 캐시 레지스터 구조

연 시간이 다른 작업에 비해 훨씬 짧기 때문에 성능에는 큰 영향을 미치지 않는다. 캐시 읽기의 경우 쓰기와 반대 순서로 작업이 진행되며, 마찬가지로 연속적인 읽기 연산에 대한 중첩 실행이 가능하다. 플래시 메모리에 대한 집적도가 증가함에 따라 셀 영역에 대한 접근 지연 시간이 증가하고 있으며, 캐시 연산은 이러한 지연 시간을 효율적으로 감추는 기능을 제공한다. 동일한 종류의 연산만이 캐시 연산 기법을 통해 중첩 실행될 수 있으므로 대량 순차 접근이 발생하는 경우에만 캐시 연산을 활용할 수 있다.

이와 같은 다중 LUN 연산, 다중 플레인 연산, 캐시 연산은 독립적인(Orthogonal) 실행 기법이기 때문에 서로 다른 기법을 혼용해서 사용하는 것이 가능하다. 예를 들어 한 LUN에 캐시 쓰기 연산을 서로 다른 플레인에 동시에 수행하는 동안 다른 LUN에는 지우기 연산을 수행할 수 있다. 현재 대부분의 상용 플래시 메모리 칩에서는 이와 같은 병렬 실행 기능들을 모두 지원하고 있다.

단일 칩 내에서 병렬 연산 처리를 지원하는 것뿐만 아니라, 복수의 칩을 병렬적으로 활용하여 성능을 향상시키는 방안도 오랫동안 연구되어 왔다. 초기에는 SSD (Solid State Drive)와 같은 고가의 대용량 저장 장치에서만 다중 칩의 병렬성 (Multi-Chip Parallelism)을 활용할 수 있었지만, MCP (Multi-Chip Package)[38] 기술이 플래시 메모리에 적용되면서 소형 휴대용 기기나 USB 메모리와 같은 저가의 소형 저장 장치에서도 다수 개의 칩을 병렬적으로 활용하는 것이 가능해졌다. MCP는 여러 개의 물리적인 칩을 적층 (Stacking)하여 하나의 패키지를 구성하는 방식으로, 하나의 칩으로 개별 패키지를 구성했던 초창기 방식에 비해 단위 면적당 용량을 획기적으로 높일 수 있기 때문에 현재 대부분의 상용 제품에서 널리 사용되고 있다.

설명한 바와 같이 단일 플래시 메모리 구조가 병렬 처리를 지원하도록 진화하고 MCP 기술이 보편화됨에 따라, 플래시 메모리가 제공하는 병렬성을 효과적으로 활용하는 기법이 더욱 중요해질 것이다. 이에 따라 불량 블록 관리 기법이나 FTL과 관련해서는 정확성을 보장하기 위해 의존성 문제를 해결하는 방안이 필수적으로 요구될 것이다.

2.5 플래시 메모리 기반 저장 장치 구조

플래시 메모리는 데이터의 덮어 쓰기 (Overwrite)가 불가능하며, 페이지 단위로 읽기 및 쓰기 연산이 수행되는데 반해 지우기 연산은 블록 단위로만 수행해야 하는 인터페이스 상의 제약이 있다. 또한 불량 블록을 허용하거나 비트 반전 오류가 발생하는 것과 같은 신뢰성 문제가 존재하며, 집적도가 증가함에 따라 플래시 메모리의 신뢰성은 점점 심각한

문제로 부각되고 있다. 신뢰성 악화를 둔화시키기 위해서 플래시 메모리 셀의 상태를 제어하는 알고리즘이 복잡해지고 있으며, 이에 따라 단일 칩의 연산 지연 시간 (Latency)이 점점 길어지고 있다. 이와 같은 인터페이스 상의 제약과 신뢰성 및 성능 저하 문제를 극복하고, 고성능/고신뢰성 저장 장치 기능을 사용자에게 제공하기 위해서 플래시 메모리를 효율적으로 관리하는 하드웨어/소프트웨어 구조 (Architecture)가 필요하다.

그림 12와 같이 전형적인 플래시 메모리 저장 장치는 하나 이상의 플래시 메모리 칩, 플래시 메모리 제어기 (Flash memory controller)[6], 장치 드라이버 (Device driver)[39][40], 플래시 변환 계층 (Flash Translation Layer; FTL)[5]으로 구성된다. 플래시 메모리가 사용되는 환경에 따라 내부 구조와 구현방식의 차이가 있을 수 있으나, 대부분 그림 12의 구성 요소를 필수적으로 구비하고 있다.

FTL은 플래시 메모리의 인터페이스 상의 제약 사항을 호스트 시스

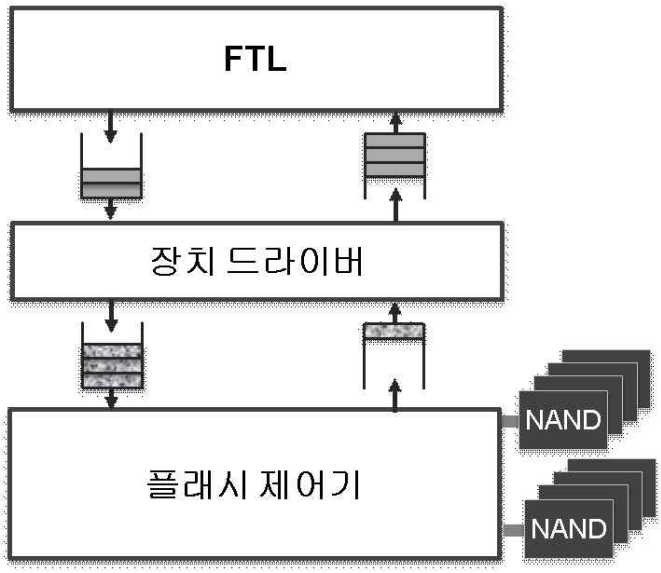


그림 12 플래시 메모리 저장 장치의 기본 구조

템에 숨기고, 하드 디스크와 같은 기존의 블록 장치 (Block device) 인터페이스를 호스트 시스템에 제공하는 소프트웨어 계층이다. 블록 장치 인터페이스는 접근의 최소 단위가 섹터 (통상 512B 또는 4KB) 인 선형적인 주소 공간을 호스트 시스템에 제공하며, 호스트 시스템은 임의 개수의 연속적인 섹터에 대하여 읽기와 쓰기 연산을 요청할 수 있다. 플래시 메모리는 덮어 쓰기를 지원하지 않고 연산의 종류 및 접근 단위가 블록 인터페이스와 상이하기 때문에, FTL은 논리적인 섹터와 물리적인 페이지 사이의 사상 (Mapping) 관계에 기반한 동적 주소 변환 (Dynamic address translation)을 통하여 덮어 쓰기가 가능한 블록 장치 인터페이스를 모사 (Emulation)한다. 호스트로부터 섹터 쓰기 요청이 발생하면 FTL은 사전에 확보해 놓은 지워진 페이지에 섹터 데이터를 쓰고, 해당 논리 섹터의 물리적인 페이지 주소를 사상 테이블 (Mapping table)에 기록한다. 호스트로부터 섹터 읽기 요청이 들어오면 사상 테이블을 참조하여 접근하려는 논리 섹터의 물리 주소를 알아내고, 이를 통해 플래시 메모리로부터 데이터를 읽어서 호스트로 반환한다. 사상 정보의 관리 기법은 FTL의 성능, 자원 사용량 및 구현 복잡도를 결정하는 핵심적인 요인이며 사상 단위 (Mapping granularity)에 따라 페이지 수준 사상 (Page-level mapping)[41][42][43]과 블록 수준 사상 (Block-level mapping)[44][45][46]으로 사상 기법을 분류할 수 있다.

페이지 수준 사상에서는 논리 섹터 주소 공간을 페이지와 동일한 크기를 가지는 논리 페이지로 나누고, 논리 페이지와 플래시 메모리의 물리 페이지 사이의 사상관계를 유지한다. 사상 단위가 작기 때문에 임의 크기의 섹터 쓰기 요청을 최소한의 페이지 쓰기 연산을 통해 수행할 수 있으며, 논리 페이지가 임의의 물리 페이지 상에 위치할 수 있으므로 다

수 개의 칩을 사용하는 경우 스트라이핑 (Striping)을 통해 쉽게 쓰기 성능을 향상시킬 수 있다. 그러나 유지해야 하는 사상 정보의 양이 많기 때문에 사상 테이블의 크기가 커지며, 이를 위한 주메모리 (Main memory) 사용량이 증가한다. 예를 들어 페이지의 크기가 4 KB이고 전체 논리 주소 공간의 크기가 64 GB인 경우, 사상 테이블의 크기는 64 MB로써[2] 스마트폰이나 이동식 저장 장치와 같이 주메모리의 크기가 제약되어 있는 환경에서는 이러한 메모리 사용량을 감당하기 어렵다. 이러한 문제점을 해결하기 위해 DFTL[47]은 플래시 메모리에 사상 테이블을 유지하고 사상 정보를 참조하거나 갱신하는 경우에 일부 사상 정보를 주메모리에 적재하는 방법을 제안하였다. 또한 자주 접근 되는 사상 정보는 가능한 오래 주메모리 상에 유지하여 사상 정보 접근의 효율성을 높였다. 사상 테이블은 페이지 크기로 분할되어 다수 개의 페이지에 기록되는데, 사상 정보가 갱신될 때마다 사상 테이블이 위치한 페이지 주소가 바뀌게 된다. 따라서 사상 테이블의 위치 정보를 유지하는 추가의 사상 테이블이 필요하며 DFTL은 이와 같은 새로운 계층의 사상 테이블을 주메모리 상에 유지한다. 만약 저장 장치 용량이 일정 한계를 넘어서게 되면 추가 계층의 사상 테이블의 크기가 지나치게 비대해져서 주메모리 용량을 초과할 수 있으며, 이는 또 다른 단계의 사상 테이블의 필요성을 의미한다. 이러한 규모 확장성 (Scalability) 문제는 저장 장치 용량이 계속하여 증가하는 한 반복적으로 발생할 것이며, HIL 프레임워크[48]에서는 사상 테이블을 관리하는 플래시 메모리 로그를 임의 계층으로 구성할 수 있게 하여 궁극적인 규모 확장성을 보장하였다.

호스트 요청을 처리하는 것과 별도로 추후 발생할 쓰기 요청을 처리

2) 사상 테이블 엔트리의 크기는 4 B로 가정하였다

하기 위해 지워진 블록을 미리 확보해 놓아야 하는데, 페이지 수준 사상 기법에서는 다수의 페이지 복사를 동반하는 쓰레기 수집 (Garbage collection) 작업[43][49][50]을 통해 빈 블록 (Free block) 을 확보한다. 호스트 쓰기 요청 시, 새로운 페이지에 데이터를 기록하면 기존 데이터가 기록되어 있던 페이지는 더 이상 사용될 필요가 없는 무효 페이지 (Invalid page)가 된다. 최신 데이터를 저장하고 있는 페이지를 유효 페이지 (Valid page)라고 하며 페이지 수준 사상에서는 한 블록 내에 무효 페이지와 유효 페이지가 혼재할 수 있다. 빈 블록을 만들기 위해서는 데이터 손실을 방지하기 위해 블록 내의 유효 페이지 데이터를 다른 블록으로 복사하여 전체 블록을 무효화한 후 지우기 연산을 수행해야 한다. 이와 같은 쓰레기 수집 작업은 데이터 복사를 위한 읽기 및 쓰기 연산을 동반하므로 저장 장치의 성능을 심각하게 저하시킬 수 있다. 따라서 복사되는 데이터의 양을 줄여서 쓰레기 수집의 효율성을 높이는 기법이 연구되었다. 또한 호스트의 요청이 존재하지 않는 유휴 시간 (Idle time)에 쓰레기 수집을 수행하도록 하여 호스트 시스템 관점의 성능 저하를 최소화하는 다양한 방법이 제안되었다.

블록 수준 사상은 사상의 단위가 지우기 연산의 단위인 블록으로써, 논리 주소 공간을 논리 블록으로 구분하고 각각을 물리 블록과 사상시킨다. 논리 블록 내의 섹터는 동일한 섹터 오프셋 (Offset)을 가지는 물리 블록 내의 섹터에 사상된다. 사상 단위가 크기 때문에 사상 정보의 양이 페이지 수준 사상 기법에 비해 현저히 작으며, 사상 테이블을 위한 주메모리 사용 부담이 줄어든다. 또한 페이지 단위로 데이터가 무효화되는 페이지 수준 사상과는 달리 블록 단위로 데이터가 무효화되므로 데이터 복사 작업 없이 빈 블록을 확보할 수 있는 장점이 있다. 그러나 논리 블

록 내의 일부 섹터만 변경하는 호스트 쓰기 요청이 발생하는 경우, 심각한 성능 저하가 발생할 수 있는데 이는 블록 단위로 데이터를 무효화시켜야하기 때문이다. 작은 크기의 덮어 쓰기를 처리하기 위해서는 동일 블록 내의 변경되지 않는 데이터들을 주메모리로 읽어온 후 쓰기 요청된 데이터와 병합하여 새로운 블록에 쓰기 연산을 수행하게 되는데, 이와 같은 일련의 과정을 읽기-변경-쓰기 (Read-Modify-Write) 작업이라고 한다. 블록 크기보다 쓰기 요청의 규모가 작거나 쓰기 요청된 섹터 영역의 경계가 논리 블록의 경계와 일치하지 않는 경우 읽기-변경-쓰기 작업이 필수적으로 동반되므로 쓰기 성능이 크게 저하되며, 플래시 메모리의 블록 크기가 확대되는 추세에 따라 이러한 문제는 점점 심각해지고 있다.

블록 수준 사상 기법의 문제점을 보완하기 위해 다양한 방식의 쓰기 버퍼링 (Write buffering) 기법이 연구되었는데, 쓰기 버퍼의 유형에 따라 휘발성 쓰기 버퍼링 (Volatile write buffering) 기법[7][51]과 비휘발성 쓰기 버퍼링 (Non-volatile write buffering) 기법[44][45][46][52]으로 구분할 수 있다. 휘발성 쓰기 버퍼링 기법은 DRAM이나 SRAM과 같은 휘발성 메모리를 버퍼로 사용하며, 호스트 쓰기 요청된 데이터를 버퍼에 우선적으로 기록하고, 버퍼 내 축적된 데이터가 일정 수준을 넘어서면 플래시 메모리에 쓰기 연산을 수행함으로써 플래시 메모리에 대한 쓰기 성능을 개선한다. 또한 버퍼 비우기 (Buffer flush) 작업 시 가능하면 동일 블록으로 향하는 데이터를 모아서 쓰기 연산을 수행하므로 읽기-변경-쓰기 작업으로 인한 성능 저하를 줄일 수 있다. 비휘발성 쓰기 버퍼링 기법은 플래시 메모리의 일부 블록을 쓰기 버퍼로 사용하며, 쓰기 버퍼 블록들은 작은 크기의 호스트 쓰기 요청을

효과적으로 저장하기 위해 페이지 수준 사상 기법을 통해 관리된다. 쓰기 버퍼 블록들이 소진되면 더 이상 호스트 쓰기 요청을 처리할 수 없으므로 블록 병합 작업을 통해 사용 가능한 쓰기 버퍼 블록을 생성한다. 블록 병합이란 블록 수준 사상으로 관리되는 데이터 블록들과 해당 블록들에 연관 (Association) 되어 있는 쓰기 버퍼 블록들로부터 최신 데이터를 읽어서 주메모리 상에 병합 (Merge)한 후 새로운 데이터 블록에 기록하는 것으로써, 기존의 데이터 블록과 쓰기 버퍼 블록을 사용 가능한 블록으로 회수하는 목적으로 사용된다. 데이터 블록과 쓰기 블록 사이의 연관 관계에 따라 블록 병합 작업의 성능 및 쓰기 버퍼 블록의 사용률 (Utilization)이 결정되는데, 하나의 데이터 블록이 하나의 쓰기 블록에 연관되는 직접 사상 (Direct-mapped) 방식[44], 복수의 데이터 블록 그룹이 복수의 쓰기 버퍼 블록 그룹과 연관 되는 집합 연관 (Set-associative) 방식[53], 임의의 데이터 블록이 임의의 쓰기 버퍼 블록과 연관될 수 있는 완전 연관 (Fully-associative) 방식[45][52][54]으로 연관 방식을 구분할 수 있다. 일반적으로 연관 방식의 자유도가 높을수록 쓰기 버퍼 블록의 사용률이 증가하여 블록 병합 횟수가 줄어들지만, 블록 병합 시 관여하는 블록의 개수가 많기 때문에 블록 병합에 소요되는 시간이 증가한다. 카멜레온 (Chameleon)[52]은 블록 병합뿐만 아니라 쓰기 버퍼 블록 영역에서 쓰레기 수집 기법을 수행하여 쓰기 버퍼 블록의 사용률 (Utilization)을 더욱 개선하였으며, LAST[54]는 쓰기 버퍼 블록 영역에서 갱신 빈도가 높은 데이터 (Hot data)와 갱신 빈도가 낮은 데이터 (Cold data)를 구분하여 서로 다른 블록에 저장함으로써 블록 병합 과정의 효율성을 향상시켰다.

호스트 요청을 처리하기 위한 사상 기법 이외에도 플래시 메모리의

오류로부터 저장 장치의 신뢰성을 보장하는 다양한 기법들도 FTL에서 수행된다. 플래시 메모리의 오류는 연산 실패로 인해 발생하는 동기적인 오류 (Internal fault)와 전원 공급 중단으로 인해 발생하는 비동기적인 오류 (External fault)로 구분할 수 있다[9][55]. 동기적인 오류에는 일시적인 비트 반전 오류와 플래시 메모리 셀의 영구적인 결함에 의한 불량 블록이 있다. 비트 반전 오류 문제는 ECC (Error Correction Code)를 통해 해결하는데, ECC는 계산 복잡도가 상당히 크기 때문에 FTL에서 수행하는 경우 저장 장치의 성능이 심각하게 저하될 수 있다. 따라서 근래의 대부분의 저장 장치에서는 ECC 기능이 내장된 플래시 메모리 제어기나 플래시 메모리 칩을 사용하여 FTL로부터 비트 반전 오류 처리 작업을 분리시켰다. 불량 블록 문제를 해결하기 위해서는, 불량 블록 발생을 최대한 늦추는 마모도 평준화 (Wear-leveling) 작업[18][19]과 불량 블록을 저장 장치 내에서 사용되지 않도록 보장하는 불량 블록 관리 (Bad block management) 기법[9][10][56]을 사용한다. 마모도 평준화는 블록 별로 지우기와 쓰기 반복 횟수를 균등하게 유지하여 특정 블록들의 수명이 조기에 소진되는 것을 방지하는 기법이다. 마모도 평준화를 통해 불량 블록 발생 시기를 늦춰서 저장 장치의 수명을 연장시킬 수 있지만, 불량 블록이 발생하는 것을 막는 것은 근본적으로 불가능하므로 블량 블록 관리 기법이 필요하다. 불량 블록이 발생하면 저장 장치의 데이터 일관성 (Data coherence)를 위해 불량 블록 내의 유효 데이터를 새로운 정상 블록으로 복사한 후, 불량 블록을 정상 블록과 교체시킨다. 교체 정보를 비롯한 불량 블록 관련 정보는 주메모리에 유지할 뿐만 아니라 불량 블록의 영구적인 배제를 위해 플래시 메모리에 비휘발적으로 저장되어야 한다. 불량 블록 관리는 사용자 데이터뿐만 아니라 별도로 발생하는 메타 정보 (Meta data)에 대한 일관성을 보장해야 하며,

불량 블록 처리 중에 중첩적으로 발생하는 내부적/외부적 오류에 대한 고려도 필요하기 때문에 본질적으로 어려운 문제이다. 게다가 다수의 플래시 메모리 연산을 병렬적으로 실행하는 환경에서는 불량 블록 처리를 위한 플래시 메모리 연산과 호스트 요청 처리와 같은 다른 작업을 위한 플래시 메모리 연산 사이의 의존성 (Dependency)으로 인해 불량 블록 처리 알고리즘의 정확성을 보장하기 더욱 어렵다. 불량 블록 관리 기법의 정확성 보장을 용이하게 하고, FTL의 구현 복잡도를 완화하기 위해 별도의 독립적인 소프트웨어 계층에서 불량 블록을 처리하는 방안이 도입되었다[9][10][39][40]. 불량 블록 관리 기법에 대해서는 2.6절에서 심도 있게 살펴본다.

초기의 FTL은 GPIO (General Purpose Input Ouput)와 같은 레지스터를 통해 입출력 신호를 제어하여 플래시 메모리 연산을 수행하였다. 이러한 방식은 플래시 메모리 칩의 성능을 효과적으로 활용할 수 없으며, 다수의 플래시 메모리 칩을 사용하는 경우 성능 저하 문제는 더욱 심각해진다. 플래시 메모리 칩의 입출력 대역폭 (Bandwidth)을 충분히 활용하기 위해서 수 ns ~ 수십 ns (nano second)의 시간 단위로 입력 신호 값을 스위칭 (Switching) 시켜야하기 때문에 대부분의 저장 장치에서는 하드웨어로 구현된 플래시 메모리 제어기를 통해 플래시 메모리에 대한 연산을 수행한다. 플래시 메모리 인터페이스가 진화함에 따라 입출력 대역폭은 증가하고 있는데, 표준화된 인터페이스 규격을 명시한 ONFI (Open NAND Flash Interface) [37]에서는 최대 400 MB/s의 속도로 명령, 주소, 데이터를 전달할 수 있는 입출력 방식을 인터페이스 상에 정의하였다. 또한 복수의 LUN과 플래인 구조 및 파이프라이닝이 가능한 페이지 레지스터 구조의 도입으로 인해 단일 플래시 메모리 칩에

서 제공하는 연산 수준의 병렬성이 증가하고 있으며, MCP (Multi-Chip Package) 기술의 보급과 더불어 단일 저장 장치에서 다수 개의 플래시 메모리 칩을 장착하는 것이 보편화됨에 따라 연산 수준의 병렬성은 더욱 풍부해지고 있다. 이와 같이 입출력 대역폭이 증가하고 병렬 연산 처리가 보편화됨에 따라 고속으로 동작하는 플래시 메모리 제어기의 중요성도 증대되고 있다.

그림 13은 플래시 메모리 제어기와 플래시 메모리 칩 간의 연결 방식을 보여준다. 입출력 핀의 개수가 플래시 메모리 제어기의 단가에 큰 영향을 미치기 때문에 입출력 핀을 공유하는 버스에 여러 개의 칩을 연결하는 방식을 사용하며, 단일 플래시 메모리 제어기는 하나 이상의 플래시 메모리 버스를 지원한다. 다수의 칩을 병렬적으로 구동시키기 위해 버스 및 칩 수준 인터리빙 (Interleaving) 기법[58]이 제안되었는데, 이는 다수의 플래시 메모리 연산을 요청된 순서대로 실행시키되 칩 충돌 (Chip conflict) 문제가 없다면 선행 연산이 완료되지 않았더라도 후행 연산을 실행시킴으로써 다수의 칩을 병렬적으로 구동하는 방법이다. 인

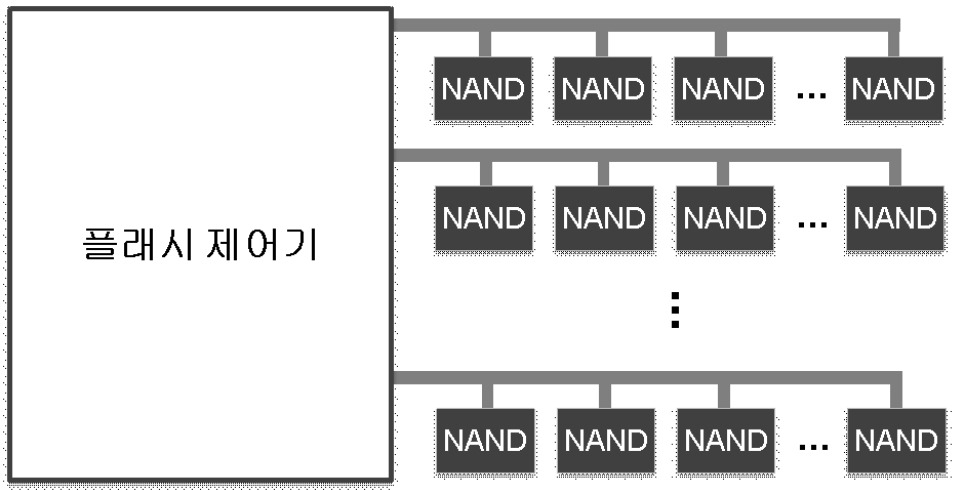


그림 13 플래시 메모리 칩의 연결 방식

터리빙 기법 하에서 다수 개의 칩이 제공하는 병렬성을 충분히 활용되기 위해서는 FTL에서 스트라이핑과 같은 방법을 통해 여러 칩으로 분산된 플래시 메모리 연산을 생성해야 하는 부담이 있다. 이를 해결하기 위해 수퍼 칩 (Super-chip)[7]57]이라는 개념이 도입되었는데, 수퍼 칩은 버스 및 칩 수준 인터리빙이 적용되는 일련의 칩들로 구성된 논리적인 칩이다. FTL이 수퍼 칩에 대한 연산 요청을 플래시 메모리 제어기로 전달하면 수퍼 칩을 구성하는 물리 칩들에 대한 인터리빙을 플래시 메모리 제어기가 자동으로 수행한다. 따라서 FTL은 용량이 크고 보다 우수한 성능의 칩을 얻은 것과 같은 효과를 얻게 되며, 더욱 편리하게 다수 칩이 제공하는 병렬성을 활용하게 된다.

인터리빙은 요청 순서에 따라 실행 시작 순서가 정해지는 고정된 연산 스케줄링 방식을 사용하기 때문에 플래시 메모리 연산들의 종류가 동일하고 접근 패턴이 규칙적일 때는 효율적으로 병렬성을 활용하지만 상이한 종류의 연산들이 불규칙한 접근 패턴으로 들어오면 효율성이 급격하게 저하된다. FTL에서 호스트 요청 처리나 쓰레기 수집과 같은 다양한 작업들이 동시에 수행되면 이질적인 연산들이 혼재되어 플래시 메모리 제어기에 요청되기 때문에, 인터리빙 기반 스케줄링 방식은 플래시 메모리 칩과 버스에서 제공하는 대역폭을 충분히 활용할 수 없다. 무순서 플래시 메모리 제어기[6]는 작업 부하 (Workload)의 특성과 상관없이 다수 개의 칩을 최대한 병렬적으로 활용하기 위해 동적 스케줄링 (Dynamic scheduling)에 기반한 무순서 중첩 실행 (Out-of-order execution) 방식을 제안하였다. 플래시 메모리 연산 간의 데이터 의존성 (Data dependency)과 오름차순 쓰기 (Ascending-order programming)에 의한 제약 이외에는 연산 순서의 제약이 없기 때문에 인터리빙 기법

에 비해 월등히 우수한 연산 처리율 (Throughput)과 응답 시간 (Response time)을 제공한다. 또한 사용자의 필요에 따라 연산 간의 우선 순위를 설정할 수 있으며, 패킷 인터페이스를 통해 모듈성, 확장성 및 재사용성을 향상시켰다. 병렬성을 최대한 활용하고 확장 가능한 인터페이스를 제공한다는 장점 때문에 본 논문에서 제시하는 불량 블록 관리 기법은 무순서 플래시 메모리 제어기의 사용을 가정하여 설계되었다.

플래시 메모리 칩을 구동하는 기능 이외에 저장된 데이터의 비트 반전 오류를 처리하기 위한 ECC 기능도 플래시 메모리 제어기에서 수행한다. ECC는 계산 복잡도가 상당히 큰 작업이기 때문에 소프트웨어로 구현하는 경우 저장 장치의 성능이 크게 저하되기 때문에 하드웨어로 구현하는 것이 바람직하며, 대부분의 상용 플래시 메모리 제어기는 자체적으로 ECC 모듈을 내장하여 FTL에 투명하게 (Transparently) 비트 오류 정정 기능을 수행한다. ECC 모듈은 ECC 부호기 (ECC encoder)와 복호기 (ECC decoder)로 구성되는데 ECC 부호기는 사용자 데이터를 기반으로 여분 정보 (Parity information)를 생성하는 역할을 하고, ECC 복호기는 여분 정보를 이용하여 데이터의 오류를 검출하고 정정하는 기능을 담당한다. 플래시 메모리에 쓰기 작업을 수행할 때마다 ECC 부호기는 여분 정보를 생성하고 이는 사용자 데이터가 쓰여지는 페이지의 예비 영역 (Spare area)에 기록된다. 읽기 연산 시 사용자 데이터와 여분 정보를 페이지로부터 읽어 내어 ECC 복호기로 전달하며, ECC 복호기는 데이터의 비트 반전 오류를 처리한 후 여분 정보를 제거하여 사용자 데이터만을 FTL로 전달한다. 비트 반전 오류의 수가 일정한 한계를 넘어가면 오류 정정 작업이 실패할 수 있으며, 이 경우 데이터가 손실되었음을 FTL에게 알려준다. 플래시 메모리를 위한 ECC로는 무작위 위치의

비트 오류 처리에 적합한 BCH 부호 (Bose-Chaudhuri-Hocquenghem code) [23][24][25]가 널리 사용되고 있으나, 플래시 메모리의 비트 오류율이 증가함에 따라 동일한 크기의 여분 정보를 가지고 더 많은 수의 비트 오류를 정정할 수 있는 LDPC 코드 (Low Density Parity Check code) [26][27]가 도입되는 추세이다.

플래시 메모리 칩의 신뢰성은 제조사마다 다르며 동일 제조사인 경우에도 제품마다 상이하다. 따라서 플래시 메모리를 위한 다양한 ECC 요구 사항이 존재하며 단일 플래시 메모리 제어기에서 모든 ECC 요구 사항을 만족시키기는 현실적으로 불가능하다. 또한 신뢰성 하락에 따라 ECC 요구 사항이 강화되면 기존의 플래시 메모리 제어기를 사용하지 못하는 하위 호환성 문제도 발생한다. 이와 같은 문제를 해결하기 위해서 최근에는 ECC 모듈이 탑재된 플래시 메모리 칩이 출시되고 있다. EZ NAND 또는 Clear NAND라고 불리는 칩들은 플래시 메모리 제어기의 복잡도를 크게 완화할 수 있고 플래시 메모리 칩의 부가 가치를 높일 수 있는 장점이 있다.

2.6 불량 블록 관리 기법

2.6.1 개요

반도체 제조 공정의 수율 (Yield)을 높이기 위해 플래시 메모리에서는 불량 블록을 허용한다. 불량 블록은 영구적인 결함이 있는 셀을 일정 개수 이상 포함하는 블록이며, 플래시 메모리 내의 블록들은 물리적으로 상호 독립되어 있으므로 불량 블록이 발생하더라도 정상 블록의 성능이

나 신뢰성은 영향 받지 않는다. 따라서 불량 블록을 적절히 관리하는 기법을 사용한다면 불량 블록이 존재하더라도 플래시 메모리를 정상적으로 사용할 수 있다. 불량 블록은 발생 원인에 따라 초기 불량 블록 (Intial bad block)과 동작 중 불량 블록 (Run-time bad block)으로 구분한다.

초기 불량 블록은 제조 공정 중 유입된 미세 먼지나 금속 불순물과 같은 오염원에 의해 발생한다. 초기 불량 블록의 판별을 위해 공장 출하 시 해당 블록의 지정된 위치에 불량임을 나타내는 값을 기록한다. 불량 블록 내의 페이지는 불량 표시를 위한 쓰기 연산이 실패할 수 있기 때문에, 불량 표시를 여러 페이지에 중복하여 기록함으로써 불량 블록에 대한 판별 실패 가능성을 낮춘다. 상용 플래시 메모리는 대부분 블록 내 처음 두 페이지에 불량 표시를 하며, 각 페이지 예비 영역의 첫 번째 바이트가 하나라도 0xFF가 아니면 불량 블록이다. 초기 불량 블록은 플래시 메모리 저장 장치가 초기화될 때 검출되어, 추후 사용되지 않도록 시스템 상에서 배제되어야 한다.

동작 중 불량 블록은 반복적인 지우기 및 쓰기 연산에 의한 마모로 인해 발생한다. 마모가 누적되어 셀에 영구적인 결함이 생기면 지우기나 쓰기에 대한 연산 오류가 발생하며, 불량 블록 관리 모듈은 이와 같은 연산 오류를 통해 불량 블록을 판별한다. 지우기나 쓰기 연산 오류의 발생 여부는 플래시 메모리 내의 상태 레지스터 (Status register)를 검사하여 확인할 수 있다.

성공적으로 쓰여진 데이터의 경우에도 쓰기 및 읽기 교란이나 데이터 유지 오류에 의해 비트 반전 오류가 발생할 수 있다. 그러나 이러한 오류 메커니즘은 정상 블록에서도 발생하는 것이며 셀에 결함을 유발하

는 것은 아니다. 따라서 비트 반전 오류가 발생한 블록이라고 하더라도 추후 지우기와 쓰기 연산을 통해 새로운 데이터가 성공적으로 기록될 수 있으며, 기존에 남아 있던 비트 반전 오류는 제거된다. 이와 같이 비트 반전 오류는 일시적인 셀의 상태 전이에 의한 것이기 때문에 불량 블록 판별 조건에 해당되지 않는다. 지우기나 쓰기 오류와 달리 비트 반전 오류는 플래시 메모리에서 자체적으로 검출하는 것이 불가능하며 통상 플래시 메모리 제어기 상에서 구현된 ECC (Error Correction Code) 회로를 통해 검출되고 정정된다.

페이지 쓰기 연산 도중 불량 블록이 발생하는 경우, 불량 블록 내에는 유효한 데이터가 남아 있을 수 있는데, 이는 쓰기 오류가 발생한 페이지가 기존에 성공적으로 쓰여진 다른 페이지에는 영향을 주지 않기 때문이다. (단, 형제 페이지 관계를 이루는 페이지는 데이터 손상이 일어날 수 있다) 데이터 손실 (Data loss)를 막기 위해 이와 같은 유효 데이터는 블량 블록 처리 중에 다른 정상 블록으로 안전하게 복사되어야 한다. 이는 단순히 교체 작업만 수행하고 별도의 데이터 보전 작업을 동반하지 않는 하드디스크의 불량 섹터 처리와 대비되는 부분이다. 또한 불량 블록을 처리할 때마다 불량 블록을 관리하기 위한 메타 정보 (Meta-data)가 갱신되며, 불량 블록에 대한 사용을 영구적으로 방지하려면 갱신된 메타 정보를 별도로 할당된 정상 블록에 기록해야 한다. 이와 같이 불량 블록 처리는 다수의 플래시 메모리 연산을 동반하는 작업이다. 불량 블록 처리 중에 지우기나 쓰기 오류로 인해 추가의 불량 블록이 발생하거나 비동기적으로 전원 오류가 발생할 수 있으며, 불량 블록 관리의 정확성을 보장하기 위해서는 이와 같은 오류 상황을 모두 고려해야 한다. 더욱이 다수의 플래시 메모리 연산이 병렬적으로 실행되는 환경에서 불량

블록 처리 중에 발생하는 연산 간 의존성 (Dependency) 문제도 고려해야 하며, 성능 저하를 최소화하기 위해 불량 블록 처리와 독립적인 연산들이 불필요하게 중단되는 것을 방지해야 한다. 이와 같은 고려 사항들은 불량 블록 관리 기법의 복잡도를 심화시키며, 정확성 보장을 매우 어렵게 한다.

불량 블록 관리 기법은 구현 방식에 따라 1) FTL (또는 파일 시스템)에 통합된 방식, 2) FTL과 하위의 장치 드라이버 계층에 불량 블록 관리 기능이 분산되어 있는 방식, 3) FTL과 독립적인 계층에서 전담하는 방식으로 구분할 수 있다. 대부분의 FTL 논문에서는 불량 블록 관리와 관련된 구체적인 내용을 다루지 않으며 상용 FTL들은 내부 구현이 공개되어 있지 않으므로 1)에 대한 관련 연구는 파악하기 어려운 실정이다. 이러한 상황은 2)의 경우에도 마찬가지이나 공개 소프트웨어인 리눅스의 FTL과 장치 드라이버를 통해 실제 구현 사례를 파악할 수 있다. 2.6.2절에서는 FTL과 장치 드라이버와 같은 하위 계층에 분산되어 구현된 불량 불록 관리 기법을 리눅스의 사례를 중심으로 설명한다. FTL과 독립적인 계층에서 전담하는 불량 블록 관리 기법은 다수의 논문과 특허를 통해 구체적인 설계 방식이 공개되어 있으며 관련 내용은 2.6.3절에서 설명한다.

2.6.2 FTL과 하위 계층에 분산된 구현 방식

리눅스의 플래시 메모리 저장 장치 서브시스템 (Subsystem)에서는 불량 블록 관리 기능을 FTL이나 파일 시스템에 독립적인 부분과 종속적인 부분으로 분할하였으며, 독립적인 부분은 MTD (Memory Technology Devices)[39]와 같은 장치 드라이버 계층에서 구현하였다.

MTD는 플래시 메모리를 위한 하드웨어 추상화 계층 (Hardware abstration layer)으로서 사용하는 플래시 메모리의 종류에 상관없이 동일한 인터페이스를 상위 계층 소프트웨어에 제공한다. NAND, OneNAND, NOR와 같은 다양한 종류의 플래시 메모리를 지원하며, 모듈화된 구조를 통해 새로운 종류의 플래시 메모리를 쉽게 수용할 수 있다.

MTD는 불량 블록 테이블 (Bad block table)을 생성하여 모든 블록의 상태 정보를 관리한다. FTL과 같은 상위 계층에서는 블록을 사용하기 전에 불량 여부를 MTD에 질의하여 불량 블록 사용을 방지할 수 있다. MTD의 불량 블록 테이블은 비트 벡터로 구현되는데, 벡터 내의 k번째 비트는 k번째 블록에 대응되며 비트 값을 설정하여 대응되는 블록의 정상/불량 여부를 표시한다. 시스템 부팅 (Booting) 시 모든 블록의 불량 표시 위치를 검사하여 주메모리 상에 불량 블록 테이블을 생성하며, 상위 계층으로부터 특정 블록의 상태에 대한 질의 요청이 들어오면 불량 블록 테이블을 참조하여 상태 정보를 반환한다. 동작 중 불량 블록은 MTD가 자체적으로 검출하지 못하며, 상위 계층에서 검출 작업을 수행하여 MTD에 불량 블록 등록을 요청해야 한다. MTD는 새로운 불량 블록을 불량 블록 테이블에 등록하고 해당 블록의 불량 표시 위치에 불량임을 나타내는 값을 기록한다. 이와 같은 불량 표시를 통해 추후 불량 블록 테이블을 재생성할 때, 동작 중 불량 블록에 대한 정보를 반영할 수 있다. 그러나 불량 블록은 신뢰성이 보장되지 않기 때문에 불량 표시 작업이 실패할 수 있으며, 현재 대부분의 플래시 메모리는 부분 쓰기 (Partial programming)를 허용하지 않으므로 쓰기 오류가 발생한 블록에 불량 표시 정보를 기록하는 것은 불가능하다.

이와 같은 문제를 해결하기 위해 MTD에서는 불량 블록 테이블을 별

도로 할당된 블록에 유지하는 기능을 제공한다. 시스템 초기화 시 처음으로 불량 블록 테이블이 생성되면, 사용자가 지정한 범위 내의 정상 블록 두 개를 선택하여 불량 블록 테이블을 기록한다. 각 블록을 원본 블록 및 복사본 블록이라고 하며 동일한 내용을 담고 있는데, 이는 데이터 중복 (Redundancy)을 통해 블록에 결함이 생기더라도 불량 블록 테이블의 소실 가능성을 줄이기 위함이다. 불량 블록 테이블이 비휘발적으로 기록되면 추후 시스템 부팅 시 전체 블록을 스캔 (Scan)하는 작업 없이 원본 블록 또는 복사본 블록을 통해 곧바로 불량 블록 테이블을 얻을 수 있다.

동작 중 불량 블록이 발생하는 경우, 플래시 메모리 상의 불량 블록 테이블을 안전하게 갱신할 수 있어야 한다. MTD에서는 전원 오류의 가능성을 고려하여 원본 블록과 복사본 블록을 순차적으로 갱신하는 방법을 사용한다. 먼저 원본 블록에 지우기 연산을 수행한 후 갱신된 블량 블록 테이블을 기록하고, 원본 블록 갱신이 완료되면 복사본 블록에도 동일한 작업을 수행한다. 전원 오류가 발생하여 원본 블록 갱신이 실패하더라도 복사본 블록에는 이전 메타 정보가 남아있으므로 기존에 등록되었던 불량 블록 정보를 복구할 수 있다. 마찬가지로 복사본 블록 갱신이 실패하더라도 원본 블록을 통해 최신 블록 정보가 복구된다. 원본 블록과 복사본 블록의 내용 불일치에 대비하여 갱신된 시점에 대한 시간 구별자 (Timestamp)도 각 블록에 기록된다. 시간 구별자는 갱신이 일어날 때마다 증가하는 값이며, 시간 구별자의 값이 큰 블록이 보다 최신의 불량 블록 테이블을 가지게 된다. 시스템이 부팅될 때, 각 블록의 시간 구별자를 읽어서 보다 큰 값을 가지는 블록으로부터 불량 블록 테이블을 복구한다.

비휘발적으로 불량 블록 테이블을 관리하는 경우에도 치명적인 결함이 있는데, 그 것은 원본 블록 및 복사본 블록을 갱신하는 과정에서 지우기 및 쓰기 오류가 발생하는 경우에 대해서 전혀 고려하지 않는다는 것이다. 예를 들어 원본 블록이 불량 블록이 되어 데이터를 저장할 수 없게 되면 불량 블록 테이블은 복사본 블록 상에만 존재하게 되며, 복사본 블록 갱신 과정 중에 전원 오류가 발생하면 블량 블록 테이블이 소실될 수 있다. 이와 같은 문제는 원본 블록 및 복사본 블록이 모두 불량 블록이 되는 경우 더욱 심각해진다. 불량 블록 테이블이 소실되는 경우에 전체 블록의 불량 표시를 스캔하여 불량 블록 테이블을 재생성하나, 이 경우 동작 중 불량 블록에 대한 정보는 여전히 복구되지 않는다.

MTD의 신뢰성 관련 결함은 MTD를 장치 드라이버로 사용하는 상위 계층에 파급된다. 리눅스의 nFTL은 MTD에서 제공받은 블록 상태 정보를 기반으로 불량 블록을 배제하는 작업을 수행하는데, MTD에 오류가 발생하는 경우 불량 블록을 정상 블록으로 오인하여 사용하게 된다. 동일한 문제는 YAFFS (Yet Another Flash File System)나 JFFS2 (Journaling Flash File System)와 같이 MTD 상에서 직접 구동되는 파일시스템에서도 발생한다. nFTL, YAFFS 및 JFFS2는 MTD로 인한 문제 이외에도 자체적인 결함이 있는데, 그 것은 지우기 오류가 발생하는 경우에만 MTD에 불량 블록 등록 요청을 수행하고 쓰기 오류는 무시한다는 것이다. 따라서 MTD가 정확히 동작하더라도 불량 블록이 배제되지 않고 계속하여 사용될 수 있으며 이는 저장 장치 시스템의 신뢰성에 치명적인 영향을 미칠 수 있다.

2.6.3 FTL과 독립적으로 구현된 방식

FTL에서 불량 블록 관리 기능의 일부 또는 전부를 담당하는 경우 저장 장치의 신뢰성을 보장하는 측면에서 다양한 문제점이 있다. 먼저 불량 블록 관리 기능이 호스트 요청 처리나 쓰레기 수집과 같은 FTL의 다양한 기능과 밀접히 연관되기 때문에 불량 블록 관리 기법의 정확성을 보장하기가 어려워진다. 또한 FTL에서는 불량 블록 관리 기능이 다양한 내부 모듈에 분산되어 구현되어 있는데, 이는 설계 상의 결함을 추적하고 수정하는데 어려움을 야기한다. 더욱이 불량 블록 처리 중에 추가의 불량 블록이 발생하거나 비동기적으로 전원 오류가 발생할 수 있으며, 이러한 중첩된 오류 상황을 모두 고려하는 경우 FTL의 복잡도가 급격하게 증가한다. 현재 중접된 오류 상황을 정확하게 처리하는 FTL 설계 기법은 공개적으로 제시되지 않았으며, FTL을 비롯한 저장 장치 소프트웨어 분야에서는 전원 오류 문제에 한정하여 정확한 처리를 보장하고 있다. 즉, 저장 매체 상의 결함이 없는 안정적인 저장 장치 (Stable storage)의 개념을 사용하여 전원 오류 복구 기법이 연구되었으며, 불량 블록과 같은 결함으로 인한 연산 오류 문제는 회피하였다.

상기의 문제점들을 해결하기 위해 장치 드라이버와 같은 별도의 소프트웨어 계층에서 불량 블록 관리 기법을 전담하는 방안이 제시되었다. 이러한 구조에서는 불량 블록 관리 기법의 정확성 보장이 용이해질 뿐만 아니라, FTL이 불량 블록을 고려할 필요가 없기 때문에 FTL의 복잡도가 크게 완화된다. 또한 FTL에 안정적인 저장 장치의 인터페이스를 제공함으로써 전원 오류가 발생하는 상황에서 FTL의 정확성을 보장하는 것이 수월해진다.

Chang 등은 불량 블록 관리 기능을 전담하도록 확장된 MTD (Reliable MTD; RMTD)[10]를 제안하였다. RMTD는 전체 블록 중 일부를 예비 블록으로 할당하며, 예비 블록 이외의 블록 (이하 데이터 블록)만을 FTL에 제공한다. 데이터 블록이 불량 블록이 되는 경우 예비 블록과 교체 작업을 수행하며, 예비 블록 중 하나를 지정하여 교체 정보를 기록한다. 추후 불량이 발생한 데이터 블록에 FTL 접근 요청이 들어오면 교체한 예비 블록으로 주소를 재지정 (Redirect) 하여 해당 요청을 수행한다. 이와 같은 예비 블록 기반의 블록 교체 기법은 FTL에 정상 블록으로만 이루어진 가상적인 플래시 메모리를 제공하기 때문에, FTL의 구현 복잡도를 완화할 수 있는 장점이 있다.

동작 중 불량 블록으로 인해 교체 정보를 갱신하는 경우, RMTD에서는 교체 포인터를 통해 전원 오류에 대비하였다. 불량 블록을 예비 블록으로 교체한 후, 불량 블록 내에 교체 포인터 값를 기록하는데, 교체 포인터는 해당 블록을 교체한 예비 블록을 가리킨다. 교체 정보 갱신 중 발생한 전원 오류로 인해 교체 정보가 소실되더라도, 모든 블록을 스캔하여 교체 포인터를 추적하면 교체 정보를 복원할 수 있다. 또한 교체 정보 블록이 불량 블록이 되면 새로운 예비 블록을 할당하여 교체 정보를 기록함으로써 중첩된 불량 블록 발생에 대비하였다. 교체 정보 블록의 위치가 동적으로 변할 수 있으므로 시스템 부팅 시 예비 블록 영역을 스캔하여 교체 정보 블록을 찾아야 한다.

기존 MTD의 문제점을 보완했음에도 불구하고 RMTD는 정확성 측면에서 몇 가지 문제가 있다. 교체 포인터는 불량 블록 내에 기록되기 때문에 교체 포인터 값 자체가 제대로 기록되거나 유지되지 않을 수 있다. 교체 포인터의 소실로 인해 해당 블록의 교체 정보가 제대로 복구되지

않을 수 있고, 이는 해당 블록의 사용자 데이터가 소실되는 결과를 초래할 수 있다. 더욱이 교체 포인터의 값이 왜곡되어 관련 없는 블록을 가리키게 되는 경우 교체 정보의 일관성을 해칠 수 있으며, 이는 다른 정상 블록 내의 데이터를 손상시키는 치명적인 문제 상황을 초래할 수 있다.

동작의 효율성 측면에도 RMTD에는 몇 가지 단점이 있다. FTL로부터 요청이 올 때마다 교체 정보를 탐색하여 블록 주소를 재지정하는데 이는 요청에 대한 응답속도를 저하시킨다. 블록 교체 과정 중에서 정상 블록을 과도하게 교체하는 문제도 존재한다. RMTD는 다중 플레인 연산을 활용하기 위해 서로 다른 플레인 내의 동일한 위치의 블록들을 묶어서 블록 집합 (Block set)을 구성한다. 실제 블록 교체는 블록 집합 단위로 수행되는데, 집합을 구성하는 블록 중 하나만이 불량 블록이 되더라도 전체 블록 집합을 교체한다. 이는 정상 블록을 불필요하게 불량 블록 처리하여 저장 장치의 수명을 단축시키게 된다. 마지막으로 교체 정보가 소실될 때마다 전체 블록을 스캔하기 때문에 시스템 부팅 시 성능이 급격히 저하되는 문제가 있다.

[59],[60]는 RMTD와 유사한 불량 블록 교체 기법을 하드웨어 형태로 구현하는 방법을 제안하였다. 하드웨어 자동화를 통해 블록 주소 재지정 작업을 가속화할 수 있으나, 확장성에 제약이 있으며, RMTD와 마찬가지로 가능한 모든 오류 상황에 대한 정확성을 보장하지 못한다.

[61]는 교체 단위의 크기를 변동하는 기법을 제안하였다. 저장 장치의 사용 초기에는 다중 플레인 연산을 활용하기 위해 블록 집합 단위로 교체를 수행하지만 교체된 블록 집합 중 정상 블록들은 추후 사용을 위

해 예비 블록 영역에 축적된다. 예비 블록 집합이 모두 소진되면, 교체 단위를 단일 블록으로 축소하고, 예비 블록을 통해 교체 작업을 수행한다. 이와 같은 기법은 RMTD에 비해 공간 활용성을 개선할 수 있지만 정확성과 성능 문제를 근본적으로 해결하지는 못한다.

[62]는 모든 블록에 대한 상태를 나타내는 비트 벡터를 통해 블록 주소 재지정에 의한 성능 저하를 감소시켰다. FTL로부터 요청이 들어오면 비트 벡터를 참조하여 대상 블록이 불량 블록인 경우에만 주소 재지정 작업을 수행한다. 불량 블록이 전체 블록에 차지하는 비율을 일반적으로 매우 작기 때문에 이와 같은 기법은 주소 재지정 횟수를 획기적으로 줄일 수 있다.

[63]는 불량 블록이 발생하였을 때 전체 블록을 교체하는 것이 아니라 오류가 발생한 페이지만을 교체함으로써 플래시 메모리 저장 장치의 수명을 연장할 수 있음을 보였다. 불량 블록을 교체용 블록 (Backing block)과 불용 블록 (Discarded block)으로 구분하였는데, 교체용 블록은 정상 페이지의 비율이 정해진 임계치보다 높기 때문에 페이지 단위의 교체를 위해 사용된다. 쓰기 오류가 발생하면 교체용 블록 공간 중의 페이지를 할당하여 쓰기 오류가 발생한 페이지와 교체한다. 교체 정보를 비롯한 메타 데이터는 테이블 형태로 관리된다. 페이지 단위의 교체 방식은 사용 가능한 저장 공간을 최대한 확보할 수 있으며 이로 인해 저장 장치의 수명을 연장할 수 있다. 그러나 동일한 블록 내의 페이지들의 마모 정도는 대부분 균일하므로 불량 블록 내의 정상 페이지는 가까운 미래에 불량이 발생할 가능성이 높다. 따라서 추후 발생할 오류를 예방하는 차원에서 블록 단위로 교체를 수행하는 것이 바람직하며, 상용 플래시 메모리 칩들은 모두 블록 단위의 교체를 요구하고 있다.

UBI (Unsorted Block Images)[40]는 MTD의 기능을 확장한 지능형 장치 드라이버로서 불량 블록 관리뿐만 아니라 마모도 평준화 및 용적 관리 (Volume management)를 수행하며 FTL에는 연산 오류가 발생하지 않는 논리적인 블록 주소 공간을 제공한다. 이를 위해 논리 블록과 물리 블록 간의 사상을 수행하며, 예비 블록 영역을 할당하여 불량 블록을 교체한다. 물리 블록의 첫 번째와 두 번째 페이지에는 사상 정보, 타임스탬프 및 지우기 횟수와 같은 메타 정보가 기록되며, 시스템 부팅 시 전체 블록을 스캔하여 메타 정보에 대한 자료 구조를 주메모리 상에 생성한다. 불량 블록이 발생하면 예비 블록을 불량 블록과 교체하는데 먼저 메타 정보를 예비 블록으로 복사한 후, 유효 데이터를 복사한다. 메타 정보 복사 시 시간 구별자의 값이 증가되어 예비 블록이 새롭게 사상된 블록임을 나타낸다.

UBI는 불량 블록 처리 시 전원 오류를 고려하지 않기 때문에 데이터 소실 문제가 발생할 수 있다. 블록 교체를 위해 메타 정보를 복사하고, 유효 데이터 복사 도중에 전원 오류가 발생하면 복사되지 않은 페이지의 데이터는 소실된다. 또한 블록 중 메타 정보를 저장하는 페이지는 FTL에서 사용하지 못하기 때문에 공간 활용 측면에서 단점이 있으며, 주메모리 상의 자료 구조를 생성하기 위해 전체 블록을 스캔해야하기 때문에 시스템 부팅 시간이 길어지는 문제가 있다. 불량 블록에 대한 교체 정보만 유지하는 RMTD와 달리 UBI는 모든 논리 블록에 대한 사상 정보가 주메모리 상에 유지되어야 하므로, 저장 장치 용량 증가에 대한 규모 확장성의 제약이 있다.

상기 설명한 기법들은 대부분 기능 확장이나 성능 향상에 중점을 두고 있으며, 정확성 보장을 위한 심도 있는 연구가 이루어지지 않았다.

특히 RMTD나 UBI와 같이 다양한 기능을 지원하는 지능형 장치 드라이버의 경우, 불량 블록 관리 기법이 다른 기능과 밀접하게 연관되어 동작하기 때문에 정확성을 보장하기가 매우 어렵다. 불량 블록 관리 기법의 결함은 저장 장치의 데이터가 손실되거나 왜곡되는 문제로 연결될 수 있으며, 메타 정보와 같이 시스템 전체에 영향을 미치는 데이터에 문제가 발생하는 경우 전체 저장 장치가 사용 불능 상태에 빠지는 치명적인 상황이 초래될 수 있다.

불량 블록 관리의 정확성을 보장하기 위해서는 플래시 메모리에서 발생 가능한 모든 오류에 대한 정확한 이해가 선행되어야 한다. 플래시 메모리 오류 모델 연구[55]은 플래시 메모리의 동작 특성을 기반으로 오류의 종류 및 속성을 체계적으로 분류하였다. 그림 14와 같이 발생 가능한 모든 오류는 발생 원인에 따라 내부적인 오류와 외부적인 오류로 구분될 수 있다. 내부적인 오류는 플래시 메모리 연산 실패에 기인하며, 동기적으로 오류가 발생하는 특성이 있는데, 이는 발생 시점이 플래시 메모리 연산과 시간적으로 연관된다는 의미이다. 내부적인 오류는 연산 종류에 따라 지우기/쓰기/읽기 오류로 세분화되며, 지우기 및 쓰기 오류는 불량 블록으로 인해 발생하므로 오류의 효과가 영구적인 속성을 가지나 읽기 오류는 일시적인 비트 반전에 의한 것이기 때문에 해당 블록을 지우고 다시 사용할 수 있는 특성이 있다. 외부적인 오류는 전원 오류와

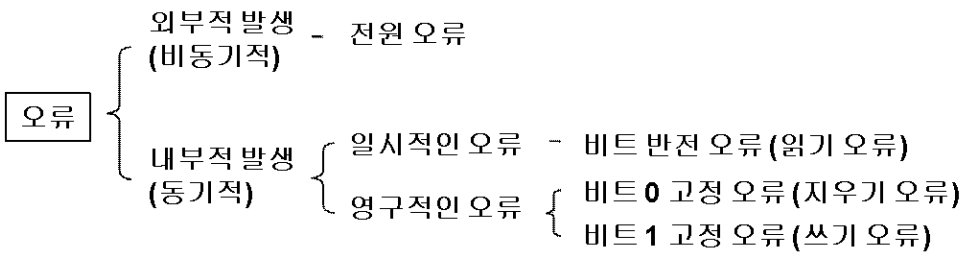


그림 14 플래시 메모리 오류의 분류

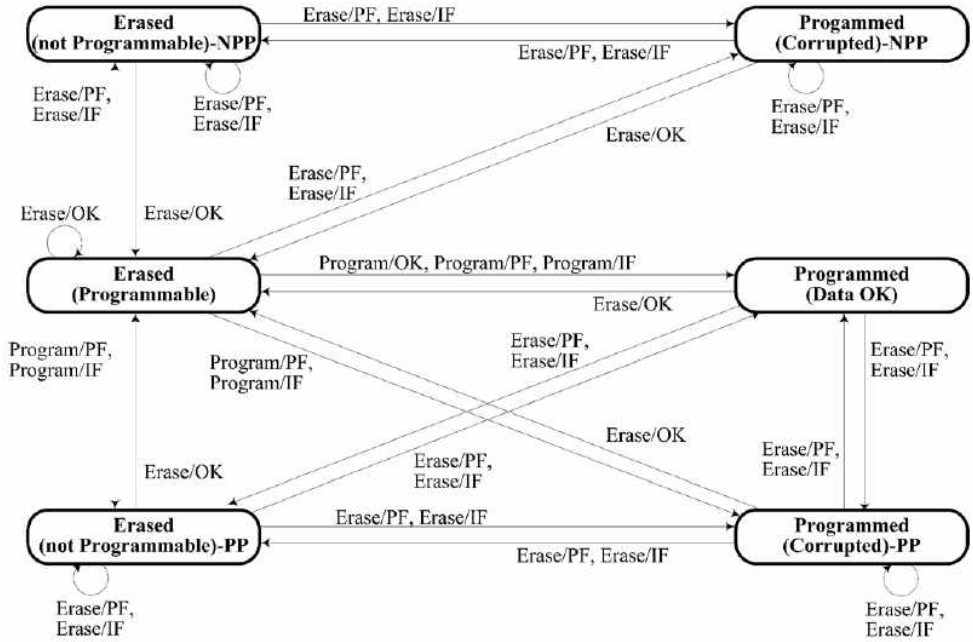


그림 15 페이지 동작 모델

같이 외부의 환경적 요인에 의해 발생하며, 플래시 메모리 연산과 시간적인 연관성이 없는 비동기적인 오류이다. 지우기나 쓰기와 같이 플래시 메모리의 상태를 변경하는 도중 전원 오류가 발생하면 해당 페이지는 오류 발생 시점에 따라 다양한 상태를 가질 수 있다. 상태 변경이 모두 완료되었거나 전혀 수행되지 않은 정상적인 상태도 가능하지만 변경이 부분적으로 완료된 상태도 가능하다. 이와 같이 부분적으로 변경된 페이지를 잔여 효과 (Residual effect) 페이지라고 하며, 잠재적으로 신뢰성 문제를 야기할 수 있기 때문에 잔여 효과 페이지에 대한 정확한 처리가 필요하다.

[55]은 플래시 메모리 연산중에 발생 가능한 모든 내부적/외부적인

오류를 고려하여, 페이지 수준의 동작 모델을 제안하였다. 그림 15의 페이지 동작 모델은 비결정적인 유한 상태 기계 (Non-deterministic finite state machine)로 표현되며, 페이지의 상태를 추상화된 값 (Erased, Data OK, Corrupted 등)으로 표현하여 상태 수를 제한함과 동시에 모델의 일반성을 확보하였다. 가능한 페이지의 상태는 다음과 같이 요약된다.

- Erased (Programmable): 해당 페이지를 포함한 블록이 성공적으로 지워졌으며 읽기 연산 시 0xFF가 읽히는 상태이다. 쓰기 연산은 이 상태에서만 요청된다고 가정한다.

- Erased (not Programmable): 불완전하게 지워졌기 때문에 성공적인 쓰기 연산이 보장되지 않으나, 읽기 연산 시 0xFF가 읽히는 상태이다. 정상적으로 데이터가 읽히는 상태 (Data OK)로 전이될 가능성에 따라 Erased (not Programmable)-PP와 Erased (not Programmable)-NPP로 세분화된다. PP (Prior Program)는 마지막으로 성공한 지우기 연산 이후로 쓰기 연산 시도가 있었음을 나타내며 Data OK 상태로 비결정적인 전이가 가능하다. NPP는 Data OK로 전이가 불가능한데, 이는 무작위 데이터 (Random data)는 ECC에 의해 오류로 검출된다는 가정에 기반한다.

- Programmed (Data OK): 읽기 시 ECC 복호를 통해 최신 데이터가 읽히는 상태 (Data OK)이다. 성공적으로 쓰기 연산이 완료되면 이 상태로 전이됨이 보장된다. 오류 상황에서도 쓰기 시도가 있었다면 비결정적으로 전이가 가능하다.

- Programmed (Corrupted): 데이터가 손상되었기 때문에 읽기 시

ECC를 통해 오류가 검출되는 상태이다.

상태 간 전이는 플래시 메모리 연산의 종류 (지우기/쓰기) 와 연산 수행 결과 (성공: OK, 연산 오류: IF, 전원 오류: PF)에 따라 결정된다. 지우기 성공과 쓰기 성공의 경우에만 정상 페이지 상태인 Erased (Programmable), Programmed (Data OK)으로 전이됨이 보장되고, 그 외의 경우에는 발생 가능한 모든 상태 중 하나로 비결정적인 전이가 일어난다. 이와 같은 전이 특성으로 인해 페이지 동작 모델의 일반성을 확보할 수 있으며, 모델 상에서 오류 처리 기법의 정확성이 검증되면 실제 시스템 상에서도 정확성이 보장된다.

X-BMS[9]는 이러한 페이지 동작 모델을 기반으로 정확성 검증이 가능한 불량 블록 기법을 제안하였다. X-BMS는 불량 블록 관리만을 목적으로 하는 독립된 소프트웨어 계층이며, 블록 재사상을 통해 불량 블록을 교체함으로써 상위 계층에는 정해진 수의 정상 블록을 제공한다. 이와 같이 불량 블록 관리 기능을 플래시 저장 장치의 다른 기능과 분리하면 정확성 검증이 용이해지는 장점이 있다. 또한 내용이 해석되지 않는 무해석 블록이라는 개념을 도입하였으며, 불량 블록 발생 시 무해석 블록에만 지우기 및 쓰기 연산을 수행하여 불량 블록 처리의 원자성을 확보하고 발생 가능한 시스템 상태 수를 제한하였다. X-BMS의 정확성 증명을 위하여 UPPAAL이라는 정형 검증 도구를 사용하였으며, 검증 결과 다음과 같은 정확성 검증 기준이 만족됨을 증명하였다.

- **데이터 일관성 (Coherence):** 쓰기 연산이 성공한 페이지는 읽기 연산 시 최신 데이터를 반환한다. 지우기의 경우 0xFF를 반환한다.

- **생존성 (Liveness):** 전원 오류가 발생하지 않거나 X-BMS의 종료 조

건에 도달하지 않는 한 상위 계층의 요청은 결국에는 성공적으로 종료된다. 단, X-BMS의 종료 조건은 무해석 블록이 모두 소진되어 불량 블록을 더 이상 처리할 수 없는 상태를 지칭한다.

- 불량 블록에 대한 무결성(Bad block integrity): 정상 블록을 불량 블록으로 오인하지 않으며, 비휘발적으로 등록된 불량 블록을 정상 블록으로 오인하지 않는다.

- 상호 배제와 전체 포괄에 대한 무결성 (MECE: Mutually Exclusive and Collectively Exhaustive integrity): X-BMS가 관리하는 블록 집합들은 상호 교집합이 존재하지 않으며, 블록 집합들을 모두 합하면 전체 블록 집합이 된다. X-BMS라는 특정 불량 블록 관리 기법에 종속적인 정확성 검증 기준이다.

X-BMS는 발생 가능한 모든 오류 상황에 대하여 불량 블록 처리의 정확성을 보장하지만, 병렬 실행에 대해 고려하지 않으며 FTL 요청 처리 시 추가의 소프트웨어 작업이 동반되기 때문에 저장 장치의 성능을 심각하게 저하시킬 수 있다. X-BMS에서 병렬 실행을 허용하는 경우 연산 간의 의존성 문제가 발생하는데, 이는 플래시 메모리 칩의 동기적/비동기적 오류와는 다른 새로운 유형의 정확성 문제를 야기한다. 본 연구에서는 X-BMS의 핵심 알고리즘을 기반으로, 병렬 처리 시에도 정확성을 보장하면서 다수의 플래시 메모리 칩에서 제공하는 대역폭(Bandwidth)을 최대한 활용하는 확장된 불량 블록 관리 기법을 제안한다. 또한 자주 사용되는 불량 블록 관리 기능을 특화된 하드웨어가 처리하게 함으로써 요청 처리 지연으로 인한 성능 저하 문제를 해결한다. 제안하는 불량 블록 관리 기법은 신뢰성과 성능을 모두 보장하기 때문에 서버용 SSD를 포함한 광범위한 저장장치 응용 제품에 적용될 수 있다.

제 3 장　의존성 문제[3] 및 해결 방안

본 장에서는 다수의 FTL 요청이 병렬적으로 처리되는 환경에서 불량 블록 처리 시 발생 가능한 의존성 문제를 규명하고, 이에 대한 해결 방안을 모색한다.

불량 블록 관리 기법 (Bad block Management Scheme; 이하 BMS)에서의 의존성 문제는 BMS 구현과 밀접하게 연관된다. 저장 장치에서의 의존성은 CPU에서의 데이터 의존성과 같이 접근하는 데이터 범위의 중첩 및 갱신 여부에 의해 결정되기 때문이다. BMS는 불량 블록 관리를 위해 다양한 메타 정보를 주메모리와 플래시 메모리 상에 유지하고, FTL 요청을 처리하거나 불량 블록을 교체하는 과정에서 메타 정보를 읽거나 갱신한다. 이와 같은 메타 정보 접근에 의해, 기존의 플래시 메모리 연산에서는 발생하지 않았던 새로운 종류의 의존성이 생길 수 있다. 또한 불량 블록을 교체하는 과정에서 데이터 일관성 (Data coherence)을 보장하기 위해 다수의 블록이나 페이지에 접근하게 되며 이로 인한 추가의 의존성이 발생하게 된다. 구체적인 의존성의 양상은 접근하는 메타 정보와 플래시 메모리 페이지의 범위 및 내용 갱신 여부에 의해 결정될 것이며 이는 BMS 구현에 종속적인 특성이 있다. 이러한 특성은

---

3) 데이터베이스나 파일 시스템 분야에서는 의존성 대신에 충돌 (Conflict)라는 용어를 사용한다. 충돌은 서로 다른 요청들이 동일 데이터에 접근하고 적어도 하나의 요청이 데이터를 갱신하는 경우에 발생한다. 그러나 플래시 메모리에서는 블록 내 오름차순 쓰기 제약으로 인해 충돌 이외에도 FTL 요청 간에 처리 순서 제약이 생긴다. 이와 같은 이유로 본 논문에서는 충돌 대신에 의존성이라는 용어를 사용하기로 한다.

의존성 문제를 규명하고 해결 방안을 제시하는데 어려움을 야기한다.

BMS 구현에 대한 일정한 가정을 도입함으로써 이와 같은 어려움을 타개하는 데에 도움이 될 수 있다. 2.6 절의 선행 조사 분석 결과에 따르면 BMS에서 블록을 관리하는 방식이나, 불량 블록을 교체하는 방법에는 공통적인 속성이 있다. 이와 같은 관찰을 바탕으로 불량 블록 처리 시에 접근하는 메타 정보의 종류 및 범위, 플래시 메모리 페이지의 범위를 한정한다면 의존성 문제를 규명하는 것이 가능하다. 이 때 중요한 고려 사항으로 적절한 수준의 가정을 도입하는 것이다. 너무 구체적인 가정을 도입한다면 가정을 만족시키지 못하는 다른 BMS 구현에 대해서는 수립한 의존성 해결 방안을 적용할 수 없을 것이다. 너무 일반적인 가정은 효율적인 의존성 해결 방안의 수립을 불가능하게 만들 수 있다. 예를 들면 불량 블록 교체 중에 임의의 메타 정보와 페이지가 갱신될 수 있다고 가정한다면, 불량 블록 교체 작업은 임의의 FTL 요청과 의존성이 발생하므로 병렬 처리가 불가능할 것이다. 따라서 기존 BMS의 구현을 충분히 포함할 수 있도록 적절한 가정을 도입하는 것이 중요하다.

3.1절에서는 이러한 관점에 따라 BMS에 대한 가정을 도입한다. 가정 도입 시 X-BMS라는 BMS 구현 기법을 중점적으로 고려하였는데, 이는 X-BMS의 정확성이 정형적으로 검증되었기 때문이다. 3.2절에서는 가정에 기반하여 의존성 문제를 규명한다. 3.3절에서는 FTL과 같은 상위 계층 및 플래시 제어기의 동작 특성을 고려하여 효율적인 의존성 해결 방안을 제시한다.

3.1 불량 블록 관리 기법 (BMS)에 대한 가정

2.6절의 관련 연구 조사에 따르면 BMS의 구현 방식은 FTL과 같은

저장 장치의 다른 구성 요소와 밀접하게 연관되는 구현 방식과 독립적인 계층으로서의 구현 방식으로 분류할 수 있다. 이 중 독립적인 구현 방식은 BMS 동작의 정확성을 검증하기 쉽고, FTL의 복잡도를 완화할 수 있는 장점이 있다. 또한 X-BMS와 같이 (의존성 문제가 없는 환경에서) 정확성이 증명된 구체적인 구현이 존재한다. 따라서 독립적인 계층으로서의 BMS를 가정한다.

이 때, BMS는 FTL과 플래시 메모리 제어기 사이에 위치하며 FTL에 불량 블록이 존재하지 않는 가상적인 블록 집합을 제공한다. 이러한 블록들을 실제 플래시 메모리의 물리적인 블록들과 구분하기 위해 의사 물리 블록이라고 명명한다. 의사 물리 블록은 지우기 및 쓰기 오류가 발생하지 않는 것을 제외하면 물리 블록과 동일한 기능과 인터페이스를 제공한다. 현재까지의 내용을 정리하면 다음과 같다.

BMS 가정 1: FTL과 독립적인 계층으로 구현되며, FTL과 플래시 메모리 제어기 사이에 위치한다.

BMS 가정 2: FTL에 불량 블록이 존재하지 않는 의사 물리 블록의 인터페이스를 제공한다.

이제 BMS가 관리하는 메타 정보 및 동작 방식에 대한 가정을 수립한다. 그림 16은 BMS가 의사 물리 블록의 인터페이스를 제공하기 위한 메커니즘을 구체적인 예(X-BMS 또는 RMTD)를 통해 보여준다. 의사 물리 블록들은 정상 블록과 일대일로 사상되며, 사상된 물리 블록이 불량 블록이 발생하는 경우 재사상을 통해 다른 정상 블록으로 사상되고 기존의 사상되었던 블록은 더 이상 사용하지 않는다. 이와 같은 재사상 작업을 위해 별도로 예비된 정상 블록들이 존재한다. 의사 물리 블록과

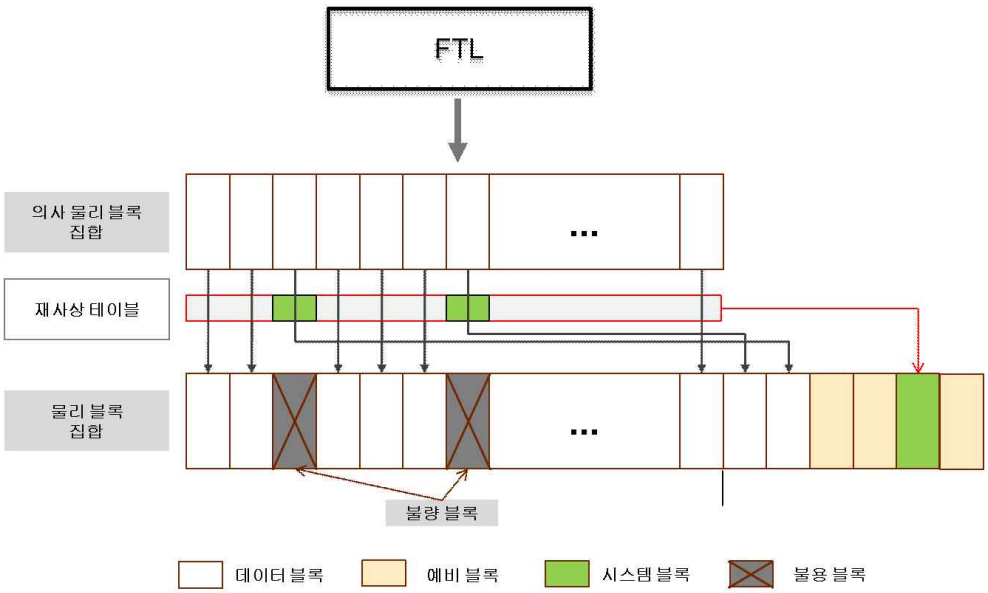


그림 16 X-BMS 및 RMTD에서의 블록 수준 사상 방식

물리 블록의 사상 정보를 관리하기 위해 사상 테이블이 존재하며, 사상 테이블은 빠른 접근을 위해 주메모리에 유지되며 전원 오류에도 소실을 방지하기 위하여 물리 블록을 별도로 할당하여 기록한다. 재사상 작업이 일어나면 사상 테이블의 내용이 갱신될 것이며, 갱신된 사상 테이블의 내용은 플래시 메모리 상에 비휘발적으로 기록된다.

그림 16에서 보여 주는 사상 방식은 X-BMS나 RMTD에서 사용하는 방식으로, 초기에 항등 사상 (Identity mapping)과 같은 미리 정해진 기본 사상을 통하여 블록 사상을 수행하고, 사상된 물리 블록이 불량 블록이 되는 경우 예비된 정상 블록으로 재사상하는 방식이다. 또한 사상 테이블의 크기를 줄이기 위하여 기본 사상된 의사 물리 블록의 사상 정보는 유지하지 않고, 재사상 정보만을 유지하고 있다. 전체에서 불량 블록이 차지하는 비율이 매우 작다는 사실을 감안하면, 이와 같은 방식은 자

원 사용 측면에서 상당히 유리할 것이다. 그러나 구체적인 사상 방식과 사상 테이블 관리 방식을 가정하는 것은 BMS 구현에 대한 과도한 제약을 부과하므로 적절하지 않다. UBI와 같이 확장된 기능을 제공하기 위해 기본 사상을 사용하지 않는 구현을 배제할 것이기 때문이다. 따라서 사상 방식에 대해서는 다음과 같은 최소한의 가정만을 도입한다.

BMS 가정 3: BMS는 의사 물리 블록과 물리 블록을 일대일로 사상하며, 사상 테이블을 비휘발적으로 유지한다.

BMS는 재사상 정보 이외의 물리 블록을 관리하기 위한 다양한 메타 정보를 유지할 것이다. X-BMS나 RMTD는 다음과 같이 물리 블록들을 구분하여 관리하며, 향후 논의의 편의성을 위해 구체적인 명칭 및 용도를 살펴 본다.

- **데이터 블록:** 의사 물리 블록에 사상되어 있는 블록으로서 FTL과 같은 상위 계층의 데이터를 저장한다.

- **예비 블록:** 불량 블록을 교체하거나 BMS의 메타 정보를 갱신하기 위해 예비된 블록이다.

- **불용 블록:** 불량 블록으로 등록되어서 더 이상 사용되지 않는 블록이다.

- **시스템 블록:** 메타 정보를 비휘발적으로 저장하기 위한 블록이다. 메타 정보에는 사상 테이블 뿐만 아니라 모든 물리 블록의 용도가 기록된다. 그 밖에 구현에 따라 추가 정보가 기록될 수 있는데, X-BMS의 경우, 부팅 시 최신의 시스템 블록의 위치를 찾을 수 있는 서명 (Signature) 정보 및 시간 구별자 (Timestamp)가 함께 기록된다.

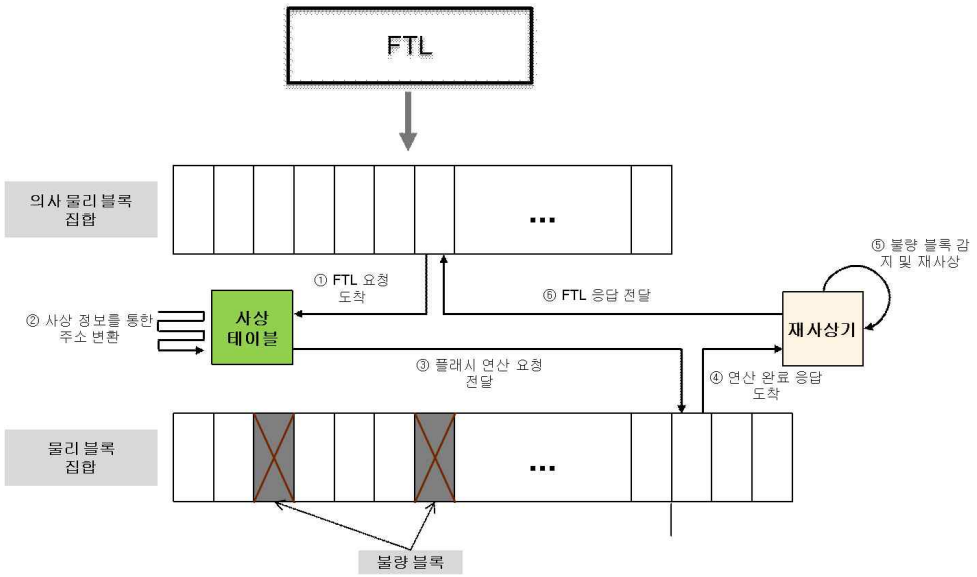


그림 17 FTL요청 처리 과정

이제 FTL 요청을 처리하는 과정을 설명한다. 그림 17과 같이 FTL 요청이 들어오면 사상 테이블을 검색하여 요청 내의 의사 물리 블록 주소를 물리 블록 주소로 변환하며, 플래시 제어기를 통해 플래시 메모리로 연산 요청을 전달한다. 연산 실행이 끝나면 요청에 대한 완료 응답이 BMS로 전달되는데, 연산 실행 중에 불량 블록이 발생하지 않은 정상 동작 상황에서는 추가의 처리 작업 없이 완료 응답이 상위 계층으로 전달된다. 지우기 및 쓰기 오류로 인한 불량 블록이 발생한 경우 재사상 작업을 수행하는데, BMS의 블록 재사상 알고리즘에서는 단순히 불량 블록을 정상 블록으로 교체하는 작업만 하는 것이 아니라, 오류가 발생한 연산이 성공적으로 완료된 것과 동일한 상태로 의사 물리 블록의 상태를 복구하는 작업을 수행한다. 따라서 블록 재사상이 끝나면 정상적으로 연산이 수행된 것과 동일하게 완료 응답을 상위 계층으로 전달한다. 이를 기반으로 재사상 작업의 기능을 가정하면 다음과 같다.

BMS 가정 4: 의사 물리 블록 B 및 페이지 p에 대한 FTL 요청 Req를 처리하던 도중 불량 블록이 발생하였을 때, 재사상 작업 완료 결과로 (1) B는 새로운 정상 물리 블록에 사상되고 (2) B의 상태는 Req가 성공적으로 완료된 것과 동일한 상태로 복구된다.

재사상 작업 과정 중에는 다수의 메타 데이터와 물리 블록에 대한 접근이 발생할 수 있으며, 구체적인 접근 범위는 BMS 구현의 종속적이다. 특정 BMS 구현과의 종속성을 최소화하면서 효율적인 의존성 해결 방안을 도출할 수 있도록 적절히 재사상 작업에 대한 가정을 도입해야 할 것이다.

BMS는 블록 단위의 사상 방식을 사용하므로, 특정 의사 물리 블록이 재사상된다면, 동일 의사 물리 블록에 대한 요청들은 해당 재사상 작업을 유발한 요청과 의존 관계가 성립하게 된다. 반면, 다른 의사 물리 블록에 대한 요청들은 의존 관계가 성립하지 않는 것이 바람직한데, 이는 재사상 작업으로 인해 다른 요청들의 처리가 지연되는 상황을 방지하기 위함이다. 따라서 다음과 같은 가정을 도입한다.

BMS 가정 5: 의사 물리 블록 B 및 페이지 p에 대한 FTL 요청 Req를 처리하던 도중 불량 블록이 발생하였을 때, 재사상 작업 과정 중에 (1) B가 아닌 다른 의사 물리 블록에 대한 사상 정보 내용은 변경되지 않으며, (2) 다른 의사 물리 블록에 사상된 데이터 블록의 내용은 참조되거나 변경되지 않는다.

위와 같은 가정은 X-BMS나 RMTD와 같은 기존 구현에서도 실제로 성립하는 가정이다. 따라서 기존 구현을 포괄할 수 있는 일반성을 확보한다고 볼 수 있다.

그 밖에 BMS의 초기화나 부팅 시의 동작과 관련된 부분이 있으나, 해당 동작 중에는 FTL 요청이 발생하지 않으므로 의존성 문제를 규명하기 위해 별도로 고찰할 필요가 없을 것이다.

3.2 의존성 문제

FTL과 같은 상위 계층 소프트웨어는 플래시 메모리에 대한 연산들이 순차적으로 실행됨을 가정한다. 즉, 플래시 제어기나 BMS와 같은 하위 계층은 요청된 순서에 따라 플래시 메모리 연산을 실행하며 이전 연산이 완료된 이후에만 다음 연산에 대한 실행을 시작한다는 것이다. 다수의 플래시 메모리 연산을 병렬적으로 실행하도록 확장된 BMS가 기존 상위 계층 소프트웨어와 올바르게 연동하기 위해서는 이와 같은 가정이 설계에 반영되어야 한다. 즉, 병렬화된 BMS의 실행 결과는 순차적으로 FTL 요청을 수행한 것과 동일한 결과를 산출해야 한다.

이와 같은 조건을 만족시키기 위해 FTL 요청 간의 의존성 문제를 규명하고 합리적인 해결책을 마련해야 한다. FTL 요청 간에 의존성이 있다는 것은 두 요청의 수행 순서 및 중첩 여부에 따라 결과가 바뀔 수 있다는 것을 의미하며, 수행 방식에 일정한 제약을 도입하여 항상 올바른 실행 결과가 나오도록 해야 한다. 반면 의존성이 없는 연산들은 수행 방식에 상관없이 항상 동일한 결과를 산출하므로 임의의 순서로 중첩하여 수행하는 것이 가능하다.

BMS에서의 연산 간의 의존성은 BMS의 동작과 독립적으로 발생하는 의존성과 종속적으로 발생하는 것으로 구분된다. 전자는 플래시 제어기

나 FTL과 같이 플래시 메모리를 이용하는 모든 시스템에서 공통적으로 발생하는 것으로서 데이터 의존성과 블록 내 오름차순 쓰기 순서 제약에 의한 의존성 (이하 쓰기 순서 의존성)이 있다. 후자의 경우, 블록 재사상 작업에 의해 발생하는 의존성으로 재사상에 의한 의존성이라고 명명한다. 데이터 의존성과 쓰기 순서 의존성은 상호 배제적으로 발생하는 특성이 있다. 즉, 데이터 의존 관계에 있는 요청들 사이에는 쓰기 순서 의존성은 성립하지 않으며 반대의 경우도 마찬가지이다. 그러나 불량 블록은 모든 지우기와 쓰기 요청을 처리하는 과정에서 발생할 수 있으므로 재사상에 의한 의존성은 다른 종류의 의존성과 동시에 성립할 수 있다. 이와 같은 성질은 의존성 분류 및 해결 방안 서술의 명료성을 저해하므로, 본 논문에서는 데이터 및 쓰기 순서 의존성이 발생하지 않는 요청들에 한정하여 재사상에 의한 의존성을 규명하도록 한다. 데이터 및 쓰기 순서 의존성을 해결하는 과정에서 불량 블록을 비롯한 발생 가능한 모든 상황을 고려한다면, 해당 의존성과 결부된 재사상에 의한 의존성은 별도로 분류될 필요가 없을 것이다.

데이터 의존성은 의사 물리 블록 또는 페이지의 내용을 변경하거나 읽는 순서에 의해 생기는 의존성이다. BMS의 경우, 상위 계층 요청의 의사 물리 주소 (블록 및 페이지 번호)를 통해 데이터 의존성 발생 여부가 결정된다. 상위 계층 요청에는 지우기, 쓰기, 읽기 요청이 있으며, 지우기나 쓰기는 블록 또는 페이지의 내용을 변경시키고 읽기는 페이지에 기록된 내용을 참조한다. 두 요청 간의 데이터 의존성이 발생하기 위해서는 (1) 동일한 페이지에 대한 접근이 일어나야 하고 (2) 두 요청 중 적어도 하나는 페이지의 내용을 변경하는 연산 요청이어야 한다. 이러한 규칙에 따르면 X-BMS에서 발생 가능한 데이터 의존성은 다음과 같이

정리된다.

- 지우기와 쓰기 (또는 읽기) 요청 간의 데이터 의존성: 동일 의사 물리 블록에 접근하는 경우에 성립한다. (페이지 위치는 관계 없다)

- 쓰기와 읽기 요청 간의 데이터 의존성: 동일 의사 물리 블록 및 동일 페이지에 접근하는 경우에 성립한다.

- 지우기 요청 간의 데이터 의존성: 동일 의사 물리 블록에 접근하는 경우에 성립한다.

- 쓰기 요청 간의 데이터 의존성: 동일 의사 물리 블록 및 동일 페이지에 접근하는 경우에 성립한다.

- 그 외 요청 간의 데이터 의존성: 성립하지 않는다.

쓰기 순서 의존성은 블록 내에서는 페이지 번호가 증가하는 순서로만 쓰기 연산을 수행해야 한다는 근래의 플래시 메모리 칩의 제약에 의한 것이다. 이와 같은 제약을 지키지 않으면 쓰기 연산 시 페이지 간의 물리적인 간섭이 과도하게 증가하여 이전에 기록된 데이터가 손상될 수 있다. 따라서 동일 블록 내 서로 다른 페이지에 대한 쓰기 요청이 연속적으로 발생하였을 때, BMS는 페이지 번호가 작은 쓰기 요청을 먼저 처리해야 한다. 이와 같은 상황은 요청된 순서가 아닌 페이지 번호에 따른 실행 순서 제약을 만들게 되며 기존의 메모리 시스템이나 저장 장치에서 발생하지 않았던 순서 제약이므로 회피하는 것이 바람직하다. 실제로 오름차순 쓰기 제약은 BMS 뿐만 아니라 FTL과 같은 상위 계층에서도 인지하는 것이므로, BMS에서는 동일 블록 내에서 먼저 발생한 쓰기 요청이 페이지 번호도 작을 것이라고 가정할 수 있다. (단, 쓰기 요청 사

이에 지우기 요청이 발생하였다면 오름차순 쓰기 제약이 없기 때문에 이러한 가정은 불필요하다) 상기 가정을 반영하여 BMS의 쓰기 순서 의존성은 다음과 같이 단순화되어 정리된다.

- 쓰기 요청 간의 쓰기 순서 의존성: 동일 의사 블록 블록 내 다른 페이지에 접근하는 경우에 발생한다 (단, 상위 계층은 항상 오름차순 쓰기 제약을 고려하여 쓰기 요청을 발생시킨다)

- 그 외 요청 간의 쓰기 순서 의존성: 성립하지 않는다.

재사상에 의한 의존성은 재사상 과정 중에 BMS 메타 정보 및 다수의 플래시 메모리 페이지를 접근하기 때문에 발생한다. 의존성 규명을 위해 고려해야 하는 경우의 수를 줄이기 위해, 데이터 및 쓰기 순서 의존 관계에 있는 요청들은 제외하기로 한다. 데이터 및 쓰기 순서 의존성을 해결하는 과정에서 재사상 작업의 수행 가능성도 고려한다면, 재사상에 의한 의존성도 자연스럽게 해결될 것이므로, 상기 의존성이 발생하지 않는 경우에 대해서만 집중하는 것이 바람직하다.

데이터 및 쓰기 순서 의존성이 없는 경우는 세 가지로 분류되는데, 서로 다른 (의사 물리) 블록으로 향하는 요청, 동일 블록 내 서로 다른 페이지로의 쓰기와 읽기 요청, 동일 블록 내 임의 페이지로의 서로 다른 읽기 요청이다. 각각에 대하여 재사상에 의한 의존성을 정리하면 다음과 같다.

- 서로 다른 블록으로 향하는 요청 사이의 재사상에 의한 의존성: 두 요청들이 모두 실행 중에 불량 블록이 발생하는 경우에 성립한다. 이를 위해 각각의 요청은 지우기나 쓰기여야 한다. 하나의 요청에만 불량

블록이 발생하는 경우에는 의존성이 성립하지 않는데, 이는 BMS 가정 5에 따라 도출되는 결과이다.

- 동일 블록 내 서로 다른 페이지로 향하는 쓰기와 읽기 요청 간의 예외 처리 의존성: 쓰기 요청 실행 중에 불량 블록이 발생하는 경우에 성립한다. 재사상으로 인해 재사상 정보가 갱신되는데, 읽기 요청 시 해당 재사상 정보가 참조되기 때문이다.

- 동일 블록 내 임의 페이지로 향하는 읽기 요청 간의 예외 처리 의존성: 성립하지 않는다.

- 그 외 요청 간의 예외 처리 의존성: 데이터 및 쓰기 순서 의존성을 해결하는 과정에서 자연스럽게 해결될 수 있으므로 별도로 고려하지 않는다.

3.3 의존성 해결 방안

이 절에서는 앞서 정리한 의존성 문제에 대한 해결 방안을 제시한다. 의존성 관계에 있는 두 요청을 정확하게 처리한다는 것은 요청 수행 결과가 해당 요청들을 순차적으로 처리한 경우와 동일함을 의미한다. 이와 같은 기준을 만족시키기 위한 다양한 방안이 존재할 수 있으며, 본 논문에서는 다음과 같은 사항을 고려하여 의존성 해결 방안을 제시한다.

(1) FTL의 동작 특성을 바탕으로 자주 발생 (Common case)하는 의존성 문제와 드물게 발생 (Rare case)하는 문제를 구분한다. 드물게 발생하는 문제의 경우, 복잡도가 작으면서도 BMS의 다른 기능과 분

리된 형태의 구현을 통해 의존성 문제를 해결하도록 한다. 이는 BMS 동작의 정확성 검증에 대한 복잡도를 줄이고, 하드웨어 가속화를 통한 최적화를 용이하게 한다.

(2) 플래시 메모리 제어기의 연산 스케줄링 특성을 이용하여 자주 발생하는 의존성 문제를 효율적으로 해결하도록 한다. 대부분의 플래시 메모리는 칩 수준의 연산 스케줄링을 수행하는데, 이는 서로 다른 칩으로의 연산은 병렬적으로 실행되지만 동일 칩으로의 연산은 요청 순서에 따라 순차적으로 실행함을 의미한다. 이와 같은 경우, 물리 블록 공간에서의 데이터 의존성과 쓰기 순서 의존성이 자연스럽게 해결되며, 이러한 성질을 이용하면 의사 물리 블록 공간에서의 의존성을 해결하는 과정에서 요청 처리가 중단되는 경우를 방지할 수 있다. 무순서 플래시 메모리 제어기나 Hydra와 같이 연산의 우선순위에 따라 동일 칩에 대한 연산의 실행 순서를 변경하는 진보된 플래시 제어기의 경우에도 물리 블록 공간에서의 의존성 문제는 여전히 플래시 메모리 제어기가 해결한다.

고려 사항 (1)을 위해서는 FTL의 일반적인 동작 특성을 먼저 고찰해야 한다. FTL이나 플래시 메모리 파일 시스템과 같은 상위 계층 소프트웨어는 덮어 쓰기가 불가능한 플래시 메모리의 제약으로 인해 로그 구조로 플래시 메모리를 관리한다. 여러 개의 빈 블록 (Free block)들을 묶어서 로그를 구성하고 사용자의 데이터를 로그의 끝에 이어 쓰는 (Appending) 방식으로 기록한다. 사용자로부터 덮어 쓰기 요청이 들어오면, 요청된 데이터는 로그의 끝에 기록하고 기존 데이터가 기록된 페이지는 무효화한다. 이와 같이 무효화된 페이지가 증가하게 되면 추후 사용자의 쓰기 요청을 처리할 수 있는 빈 페이지가 줄어들게 되기 때문

에, 쓰레기 수집 (Garbage collection)이라는 작업을 수행하여 쓰기 요청을 처리하기 위한 빈 블록들을 확보한다. 쓰레기 수집은 통상 별도의 모듈에서 수행하며, 지우기 대상 블록을 선정하여 해당 블록의 유효한 데이터들을 다른 블록으로 복사하고 대상 블록을 지움으로써 빈 블록들을 생성한다. 이 때, 유효한 데이터 소실을 막기 위해서 데이터 복사가 성공적으로 완료된 후에만 지우기 작업을 수행한다. 쓰레기 작업 결과로 확보된 빈 블록은 빈 블록 리스트와 같은 자료 구조를 통해 관리되다가 쓰기 로그를 관리하는 모듈에 공급되어 사용자의 쓰기 요청을 처리하는 데 사용된다.

로그 방식의 쓰기 작업과 쓰레기 수집 작업이라는 특성 이외에도 대부분의 상위 계층 소프트웨어는 일정한 양의 휘발성 버퍼를 사용하는 공통점이 있다. 쓰기 요청된 데이터는 플래시 메모리에 기록되기 전에 DRAM과 같은 휘발성 버퍼에 임시로 저장되는데, 이는 버퍼에 데이터가 저장되자마자 쓰기 응답을 수행하여 사용자가 체감하는 응답 속도를 개선하기 위함이다. 버퍼 내의 데이터는 가장 최근에 쓰기 요청된 데이터이므로, 사용자의 읽기 요청 시 버퍼 내에 요청된 데이터가 있으면 버퍼로부터 읽기 요청을 처리하고 플래시 메모리에는 읽기 연산을 수행하지 않는다. 버퍼 내의 데이터는 저장 장치의 데이터 일관성 보장을 위해 플래시 메모리에 쓰기 작업이 완료된 이후에만 버퍼 상에서 제거된다.

이와 같은 상위 계층의 동작 특성을 바탕으로 BMS 상에서 데이터 및 쓰기 순서 의존성 문제 양상을 다음과 같이 정리하였다. (재사상에 의한 의존성은 불량 블록 발생과 관계되므로 FTL의 동작 특성과 직접적으로 연관 짓기 어렵다.)

(1) 데이터 의존성은 거의 발생하지 않을 것으로 기대된다. 실제로 HIL 과 같이 정확성을 보장하는 일반적인 FTL 설계 방법론에 의하면 BMS 상에서 데이터 의존성 문제가 전혀 발생하지 않는다. 이에 대한 근거는 다음과 같다

- 지우기와 쓰기 요청 간의 데이터 의존성: 대부분의 FTL은 지우기 작업과 쓰기 작업을 별도의 모듈에서 수행하며, 저장 장치 동작의 정확성을 위해 모듈간의 동기화를 수행한다. 쓰기 모듈은 지우기 모듈이 공급하는 빈 블록에 대해서만 쓰기 작업을 수행하기 때문에 지우기가 미완료된 블록에 대한 쓰기 요청은 발생하지 않는다. 지우기 모듈의 경우 내부에 유효한 데이터가 없는 블록에 대해서만 지우기 작업을 수행하므로, 쓰기 모듈이 쓰기 작업을 진행하는 블록에 대한 지우기 요청은 발생하지 않는다.

- 쓰기와 읽기 요청 간의 데이터 의존성: 상위 계층은 버퍼에서 플래시 메모리로 쓰기 작업이 완료된 후에만 버퍼 상에서 데이터를 제거한다. 쓰기 작업이 진행 중인 데이터는 버퍼 상에도 유지되므로 사용자의 읽기 요청은 플래시 메모리가 아닌 버퍼 상에서 처리된다. (즉, BMS로 읽기 요청이 전달되지 않는다) 읽기 요청이 진행 중인 페이지에 대한 쓰기 요청은 덮어 쓰기가 불가능한 플래시 메모리의 특성 상 발생하지 않는다.

- 지우기 요청 간의 데이터 의존성: 지우기가 진행 중인 블록에 추가의 지우기 요청을 수행하는 것은 무의미하므로, 발생 가능성을 생각하기 어렵다.

- 쓰기 요청 간의 데이터 의존성: 덮어 쓰기가 불가능한 플래시 메모

리의 특성 상 발생하지 않는다.

- 지우기와 읽기 요청 간의 데이터 의존성: 지우기가 진행 중인 블록에는 유효한 데이터가 존재하지 않으므로 읽기 요청이 발생하지 않는다. 반대로 읽기 요청이 진행 중인 블록은 유효한 데이터가 존재한다는 것을 의미하므로 지우기 요청이 발생하지 않는다.

(2) 쓰기 순서 의존성은 상위 계층의 쓰기 방식의 특성 상 자주 발생한다. 대부분의 상위 계층은 로그 형태로 의사 물리 블록들을 관리하며, 로그에 이어 쓰는 방식으로 쓰기 작업을 수행한다. 이러한 쓰기 방식에 따라, 특정 블록에 페이지 번호가 증가하는 다수 개의 쓰기 요청을 연속적으로 발생시킬 가능성이 높다. 이와 같은 상황을 고려하면, 쓰기 순서 의존성이 발생하더라도 쓰기 요청 처리가 지연되지 않도록 의존성 해결 메커니즘을 설계하는 것이 바람직하다. 플래시 메모리 제어기가 물리 연산 간의 쓰기 순서 의존성을 지키는 범위에서 연산 스케줄링을 한다는 것을 감안하면, 불량 블록이 발생하지 않는 대부분의 경우에 요청 처리의 지연 없이 쓰기 작업을 수행하는 것이 가능하다.

상기 설명한 의존성 문제의 발생 양상을 고려하여 의존성 해결 방안을 제시하도록 한다.

3.3.1 데이터 의존성 해결 방안

플래시 메모리 저장 장치에서는 병렬적으로 처리되는 요청 간의 데이터 의존성이 발생하는 상황은 극히 드물기 때문에, 데이터 의존성 해

결을 위한 구현의 복잡도를 줄이는 것이 바람직하다. BMS 상에서 의존성 문제가 불량 블록으로 인한 재사상 작업과 결합되는 경우, 의존성 해결 과정 시 고려해야 하는 경우의 수가 늘어나게 된다. 이는 구현 복잡도를 증가시키고 정확성 보장을 어렵게 하기 때문에, 불량 블록 처리와 독립된 별도의 계층에서 데이터 의존성을 해결하는 것이 합리적이다. 데이터 의존성 처리 계층은 BMS와 FTL 사이에 위치하며 데이터 의존성이 감시되었을 때, FTL 요청을 의존성이 해결될 때까지 지연함으로써, 간단하게 의존성 문제를 해결할 수 있다.

이와 같이 독립된 계층에서 데이터 의존성을 해결하는 경우, BMS에서는 데이터 의존성을 고려할 필요가 없기 때문에 구현 및 검증에 다한 부담이 부가되지 않는 장점이 있다. 데이터 의존성이 발생하는 경우 요청 처리가 지연되지만, 데이터 의존성이 발생할 가능성은 극히 작기 때문에 성능에 미치는 영향은 경미할 것이다. 실제로 대부분의 FTL은 데이터 의존성을 발생시키지 않을 것으로 기대할 수 있는데, 이는 모듈화된 구조로 FTL이 구현되고 모듈 간의 동기화를 통해 데이터 의존성을 자체적으로 해결하기 때문이다. 다양한 FTL을 구현할 수 있는 HIL 프레임워크에서는 플래시 메모리 저장 장치의 정확성을 보장하기 위해 작업 간의 동기화 규칙을 도입하였으며, 그러한 규칙에 의하면 의사 물리 블록 공간에서의 데이터 의존성은 발생하지 않는다. 이와 같이 데이터 의존성을 상위 계층에서 자체적으로 해결한다면 데이터 의존성 처리 계층을 저장 장치 시스템 구조에서 제거할 수 있다. 그림 18과 같이 데이터 의존성 처리 계층은 BMS와 독립된 모듈 형태로 동작하므로 개발자는 필요에 따라 해당 계층을 쉽게 제거하거나 재사용할 수 있다.

3.3.2 쓰기 순서 의존성 해결 방안

쓰기 순서 의존성은 데이터 의존성과 달리 자주 발생하는 문제이다. 이는 특정 개수의 블록에 대해서 이어 쓰는 방식으로 쓰기 요청을 전달하는 FTL의 동작 특성에 기인한다. 쓰기 순서 의존성을 그림 18과 같이 BMS와 독립적인 계층에서 해결하는 경우, 쓰기 작업의 지연이 너무 많이 발생하여 저장 장치 성능이 저하될 것이다. 특히 캐시 쓰기 기능을 지원하는 근래의 플래시 메모리 칩과 제어기의 경우, 쓰기 작업 지연으로 인한 성능 저하는 더욱 클 것이다. 캐시 쓰기 기능은 동일 블록에 대한 연속적인 쓰기 연산을 파이프라인 구조의 내부 버퍼를 통해 중첩하여 실행하는 방식인데, FTL의 쓰기 요청 처리가 지연된다면 이러한 캐시 쓰기 기능을 활용하지 못하기 때문이다. 따라서 쓰기 순서 의존성은 BMS가 자체적으로 해결하도록 하며, 동작 중 불량 블록이 발생하지 않는 대부분의 경우에 쓰기 요청 처리가 지연되지 않도록 하는 것을 의존성 해결 메커니즘 설계의 기본적인 목표로 한다.

이러한 목표를 달성하기 위해 플래시 메모리 제어기의 동작 특성을 파악하고 이를 BMS 설계에 고려하는 것이 필요하다. 3.3절의 고려사항 (2)에서 정리한 바와 같이 플래시 메모리 제어기는 동일 물리 블록 내의 쓰기 연산에 대해서는 순차적으로 처리한다. 캐시 쓰기 기능을 활용하는 경우에는 쓰기 연산을 파이프라인 방식으로 중첩하여 실행하지만, 연산 결과는 순차 실행의 결과와 동일함이 보장된다. 이러한 특성을 이용하여 X-BMS 상에서의 쓰기 요청 처리 지연을 방지하도록 한다.

BMS의 쓰기 요청 처리 과정은 쓰기 오류 없이 정상적으로 처리되는 경우와 쓰기 오류로 인해 요청 처리 과정 중에 블록 재사상 작업을 수행하는

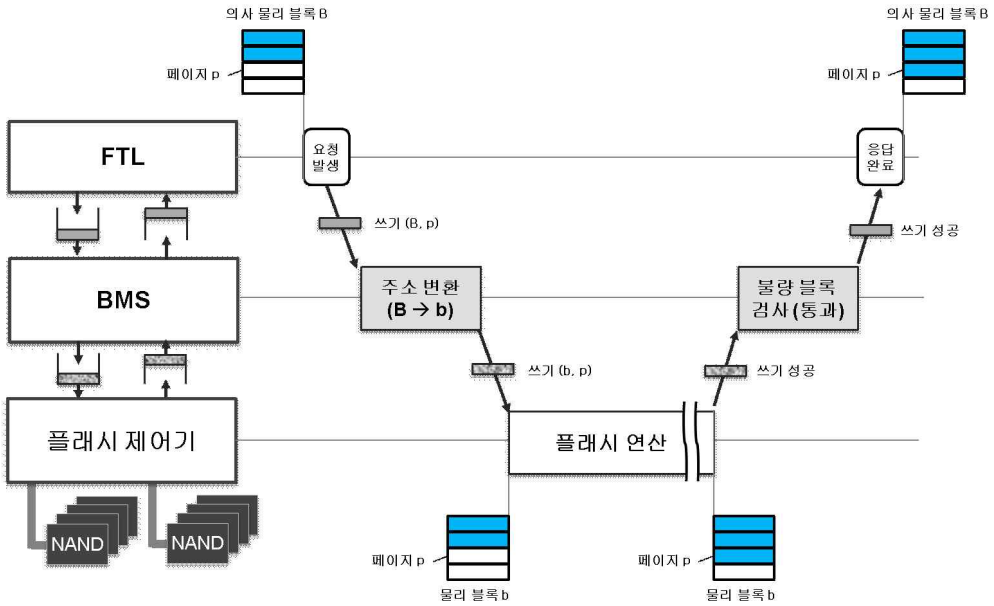


그림 18 쓰기 요청 처리 과정 (쓰기 연산이 성공한 경우)

경우로 나눌 수 있다. 대부분은 정상적으로 처리되는 경우이며 이때의 처리 과정은 그림 18과 같이 재사상 테이블을 참조하여 의사 물리 블록 주소를 물리 블록 주소로 변환하는 단계, 플래시 메모리 제어기 상에서 연산을 실행하는 단계, FTL로 실행 결과를 응답하는 단계로 나누어진다. 이제 쓰기 순서 의존 관계에 있는 두 요청을 실행하는 방법을 고찰해 보자. 의사 물리 블록 B의 연속적인 페이지 1 및 2에 대한 두 쓰기 요청에 대한 순차 실행 과정은 그림 19와 같다. 쓰기 요청 i는 재사상 정보를 참조하여 B에 사상된 물리 블록 b로 주소가 변환되었으며, 물리 주소 (b, 1)에서 정상적으로 실행된 후 상위 계층으로 응답된다. 쓰기 요청 j도 동일한 과정으로 처리되었으며 물리 주소 (b, 2)에서 실행된 후 상위 계층으로 응답되었다. 그림 20의 (a), (b), (c)는 요청 i, j과 중첩되어 실행되지만 순차 실행과 동일한 결과를 산출하는 예를 보여준다. 그림 20 (a)에서는 요청 실행의 세부 단계들이 순

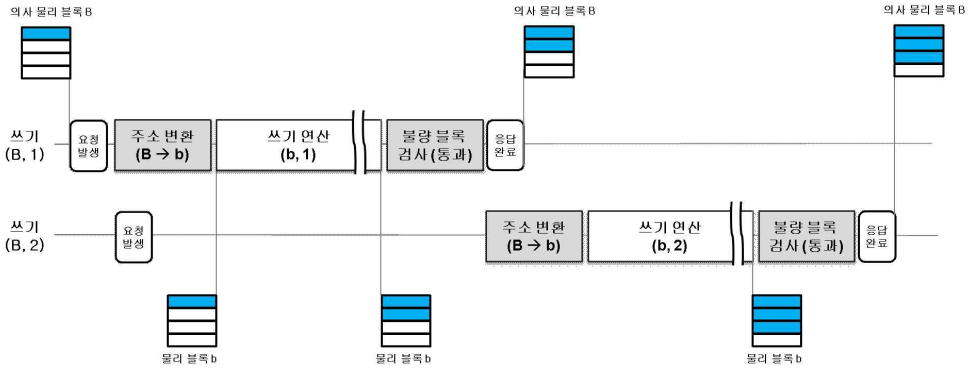


그림 19 순차적인 쓰기 요청 처리 과정 (쓰기 연산이 성공한 경우)

서대로 수행되는 경우이며, 그림 20 (b)는 재사상 정보 참조 단계의 순서가 뒤바뀐 경우, 그림 20 (c)는 응답 단계도 뒤바뀐 경우를 보여준다. 세 경우 모두 순차 실행과 동일한 실행 결과를 얻게 되는데, 이는 상태를 변경하는 플래시 연산 실행 단계가 순차적으로 처리되기 때문이다. 따라서 동작 중 불량 블록이 발생하지 않는 경우에는 플래시 메모리 제어기로의 연산 실행 요청 순서만 지켜준다면 쓰기 순서 의존성 문제는 자동적으로 해결된다. 그림 20 (d)는 연산 실행 단계의 순서가 뒤바뀐 경우이며, 물리 블록에 대한 개별 쓰기 연산은 성공적으로 수행되었지만 오름차순 쓰기 순서 제약을 위반하였기 때문에 페이지 2에 기록된 데이터가 페이지 1에 대한 쓰기 연산으로 인해 손상되었다.

그림 20 (c)는 인터페이스 측면에서는 순차 실행과 차이점이 있는데, 이는 요청된 순서와 응답 순서가 다르다는 것이다. 이러한 성질은 FTL에서 저장 장치 일관성을 위해 순차적인 응답을 요구하는 경우 시스템 전체적인 측면에서 정확성 문제를 야기할 수 있다. 따라서 X-BMS의 일반성을 확보하기 위해 상위 계층으로의 응답은 순차적으로 수행하는 것이 바람직하다.

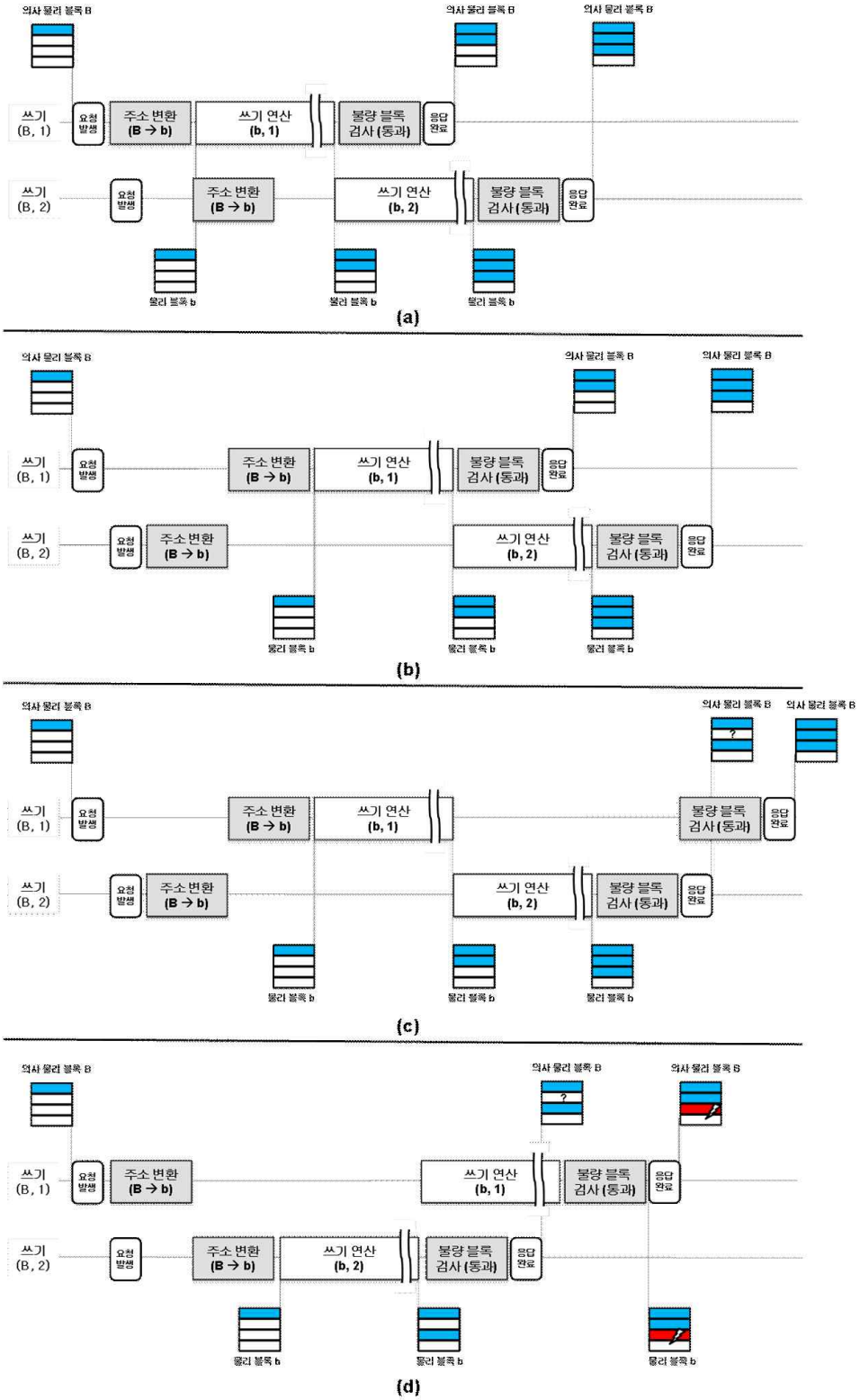


그림 20 중첩된 쓰기 요청 처리 (쓰기 연산이 성공한 경우)

이와 같은 제약은 쓰기 순서 의존 관계에 있는 요청 사이에 국한되는 것은 아니며, 모든 FTL 요청 처리 과정에서 적용되는 것이다. 무순서 플래시 메모리 제어기의 경우, 내부적으로 재정렬 버퍼 (Reorder buffer)를 구비하여 연산 완료 응답을 순차적으로 수행하는데, 이러한 진보된 플래시 메모리 제어기를 사용한다면 상기 제약으로 인해 BMS에 부과되는 부담은 경미할 것이다.

지금까지 불량 블록이 발생하지 않는 경우의 처리 과정을 살펴보았다. 이때의 쓰기 순서 의존성 해결을 위한 규칙을 정리하면 다음과 같다.

실행 규칙 1: 쓰기 순서 의존 관계에 있는 두 FTL 요청에 대하여, 플래시 메모리 제어기로 연산 실행 요청은 순차적으로 전달한다.

또한 순차 실행과 인터페이스 측면의 동일성을 보장하기 위하여 모든 상위 계층 요청에 대하여 다음과 같은 규칙을 부과하였다.

응답 규칙: 모든 요청 처리에 대한 완료 응답은 FTL에 순차적으로 전달한다.

이제 쓰기 연산 중 불량 블록이 발생하는 경우에 대해서 살펴본다. 불량 블록이 발생한 경우의 요청 처리 과정은 그림 21과 같으며, 크게 의사 물리 블록 주소를 물리 블록 주소로 변환하는 단계, 플래시 메모리 제어기 상에서 연산을 실행하는 단계, 불량 블록으로 인해 재사상을 수행하는 단계, FTL로 실행 결과를 응답하는 단계로 나누어진다.

의사 물리 블록 B의 페이지 p에 대한 쓰기 요청에 대한 요청이 들어오면, 주소 변환 과정을 통해 B는 물리 블록 b1으로 변환되어 플래시 제어기로 전달된다. b1에서 쓰기 연산 중 오류에 의해 불량 블록이 발생하였고,

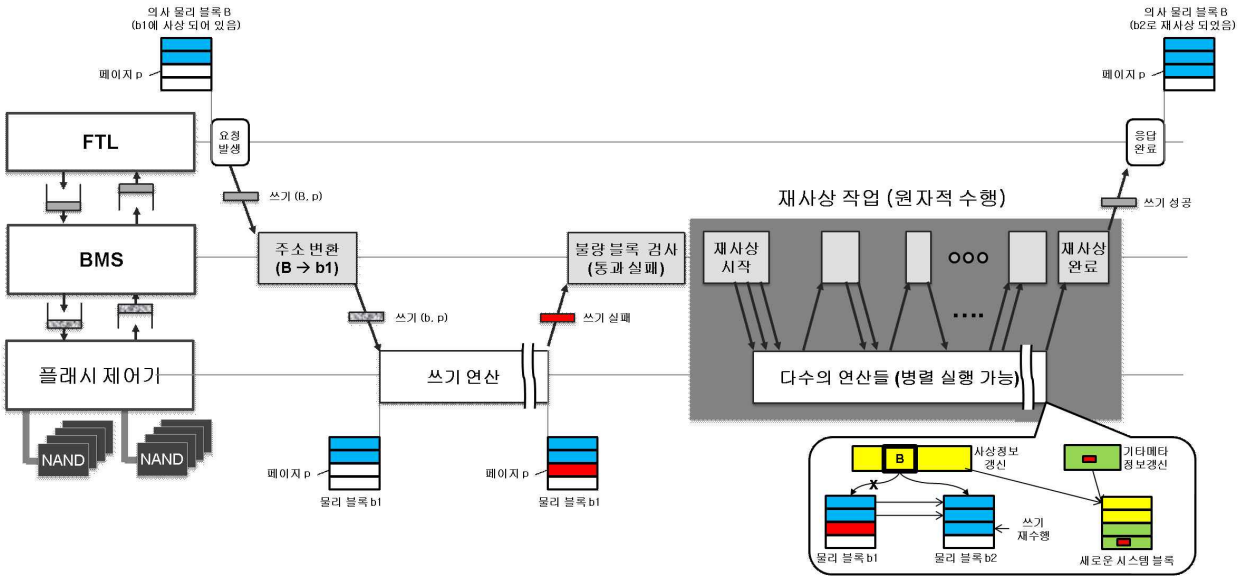


그림 21 FTL 쓰기 요청 처리 과정 (쓰기 오류가 발생한 경우)

이를 복구하기 위해 재사상 작업이 수행된다. 재사상 작업은 b1내의 유효 데이터 복사, 실패한 쓰기 연산 재수행, 메타 정보 기록 등을 위해 다수의 플래시 메모리 연산들을 동반하게 된다. 재사상 작업의 결과로, B는 정상 블록 b2에 사상되며, b2의 상태는 쓰기 요청 (B, p)가 성공적으로 완료된 상태와 동일하므로, 재사상 완료와 함께 FTL로 성공적인 쓰기 응답이 전송된다. 이 후 B에 대해 발생한 요청들은 주소 변환 과정에 의해 b2 상에서 실행될 것이다.

불량 블록 재사상 단계는 다수의 세부 단계로 이루어지며 각 단계 별로 메타 정보 및 플래시 메모리의 내용을 변경한다. 이러한 세부 단계 별 내용 갱신을 고려하여 의존성을 해결하는 경우 문제의 복잡도가 급격히 증가할 뿐만 아니라, 재사상 알고리즘을 변경하는 경우에 기존에 수립된 의존성 해결 규칙을 사용하지 못하게 된다. 따라서 재사상 작업 진행 중에는 임의의

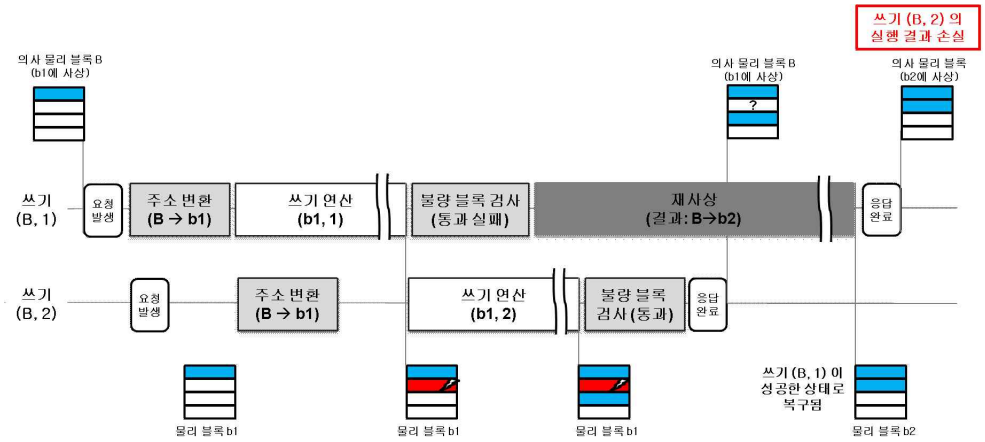


그림 22 재사상으로 인한 쓰기 손실 문제

시점에 메타 정보 및 플래시 메모리의 내용이 갱신된다고 가정하며, 이에 따라 갱신 중인 정보를 참조하는 경우 참조 결과의 일관성을 보장할 수 없다. 그림 21의 예에서 재사상 정보 B->b1을 B->b2로 갱신하는 도중에, B에 대한 다른 쓰기 요청이 발생하여 재사상 정보를 참조하게 되면, 참조 결과로 반환되는 값은 b1, b2 모두 가능하며, b1을 반환하는 경우 최신의 재사상 정보가 아니므로 쓰기 요청 수행 결과에 오류가 생길 것이다.

그림 22는 재사상 정보 참조의 일관성이 지켜지지 않았을 때, FTL에 미치는 효과를 보여준다. 동일 의사 물리 블록에 대한 쓰기 요청 (B, 1)과 (B,2)가 연속적으로 BMS에 도착하였으며, 실행 규칙 1에 따라 쓰기 요청 (B, 1)이 물리 주소 (b1, 1)로 전달되고, 이후 쓰기 요청 (B, 2)가 물리 주소 (b1, 2)로 전달되었다. (b1, 1)에 대한 쓰기 연산 도중 오류가 발생하여 재사상 작업이 수행되지만, 쓰기 연산 (b1, 2)는 재사상 이전에 플래시 메모리로 전달되었으므로 불량 블록인 b1에서 쓰기 작업을 수행한다. 쓰기 연산 (b1, 2)가 성공적으로 끝나더라도 쓰기 연산 결과는 의사 물리 블록 B 상에

서 손실될 것인데, 재사상 작업 완료와 함께 B가 b1에서 b2로 사상될 것이기 때문이다. 재사상 작업은 쓰기 요청 (B, 1)이 성공적으로 수행된 것과 같은 상태로 b2를 복구하므로, 쓰기 요청 (B, 2)의 결과는 사라지게 된다.

이러한 쓰기 손실 (Lost update) 문제를 방지하기 위해서는 모든 쓰기 요청이 주소 변환 과정 중에 최신의 사상 정보를 참조하도록 해야 한다. 이를 보장하기 위해서는 쓰기 요청 처리 중 재사상 작업이 동반될 때, 의존 관계에 있는 후행 쓰기 요청들은 재사상 작업이 완료된 후에만 재사상 정보를 참조해야 한다.

그러나 이와 같은 방법에도 한계가 있는데, 후행 요청의 도착 시점에 따라 선행 요청의 재사상 작업 수행 여부를 사전에 알 수 없기 때문이다. 그림 22에서와 같이 쓰기 (B, 2)는 쓰기 (B, 1)을 위한 재사상이 시작하기 전에 도착했으므로, 쓰기 (B, 1)의 쓰기 연산 결과가 나올 때까지 재사상 수행 여부를 알 수 없다. 이 경우, 쓰기 오류 여부를 알 수 있을 때까지 쓰기 (B, 2)의 처리를 지연시키는 것이 보수적인 방법이나 성능 저하를 유발할 수 있다. 불량 블록으로 인해 재사상 작업을 수행하는 것은 매우 드문 경우이기 때문에, 대부분은 불필요한 처리 지연이기 때문이다.

이러한 문제를 해결하기 위해서 사후에 쓰기 손실을 판단하여, 쓰기 연산을 재수행하는 방법을 사용할 수 있다. 그림 23은 그림 22의 예에서 쓰기 손실을 판단하여 연산을 재수행하는 과정을 보여준다. 쓰기 (B, 2)는 선행 쓰기 (B, 1)의 결과를 기다리지 않고 처리가 시작되며, 불량 블록이 된 b1에서 수행되었다. 그러나 쓰기 (B, 2)를 위한 연산 완료 시점에서 선행 쓰기 (B, 1)으로 인한 재사상 여부를 알 수 있으므로, 기존의 수행된 작업을 무효화하고 재사상 완료된 후에 처리를 재시작할 수 있다. 쓰기 (B, 2)로 인한

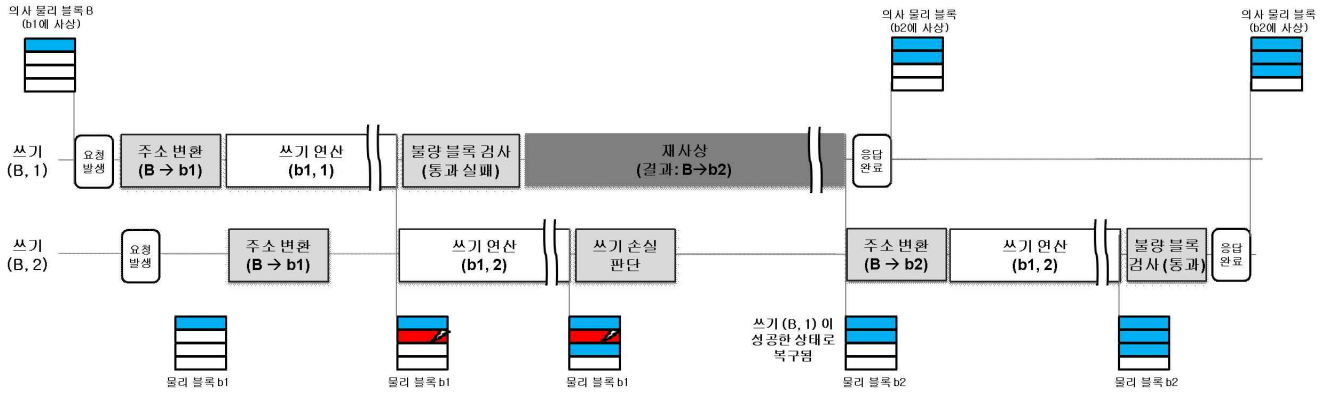


그림 23 작업 무효화를 통한 쓰기 손실 해결

효과는 불량 블록 b1에 남아 있지만, BMS에서는 불량 블록을 더 이상 사용하지 않기 때문에 정확성에는 영향을 끼치지 않는다.

지금까지 선행 쓰기 요청 처리 시 불량 블록이 발생하는 경우를 살펴보았다. 이 때의 쓰기 순서 의존성 해결을 위한 규칙을 정리하면 다음과 같다.

실행 규칙2: 쓰기 순서 의존 관계에 있는 두 FTL 요청에 대하여, 선행 요청을 위한 재사상 작업이 시작된 후에 후행 요청이 도착하였다면, 후행 요청은 재사상 작업이 완료된 이후에만 처리가 시작되어야 한다.

실행 규칙3: 쓰기 순서 의존 관계에 있는 두 FTL 요청에 대하여, 선행 요청을 위한 재사상 작업이 시작하기 전에 후행 요청이 도착하여 처리가 시작되었다면, 후행 요청의 기존 작업은 무효화되고 재사상 작업이 완료된 이후에 재수행되어야 한다.

후행 쓰기 요청 처리 시에 불량 블록이 발생하는 경우를 위한 별도의 규칙은 필요하지 않은데, 그 이유는 실행 규칙 1, 2, 3을 지키는 경우 후행 요청의 재사상 수행 여부는 쓰기 의존 관계에 있는 선행 요청의 수행 결과에

영향을 미치지 않기 때문이다. 규칙 1, 2, 3에 의하면 후행 요청의 (무효화되지 않은) 쓰기 연산이 완료되고 재사상 작업이 시작되는 시점은 선행 요청의 쓰기 연산 및 재사상 작업이 모두 완료된 시점이기 때문에, 불량 블록이 발생하더라도 선행 요청의 결과에 영향을 미치지 않는다.

3.3.3 재사상에 의한 의존성 해결 방안

데이터 의존성이나 쓰기 순서 의존성이 성립하지 않는 요청 간에도 요청 처리 중에 불량 블록이 발생하는 경우 의존성이 발생할 수 있다. 3.2.1절에서 정리한 바에 따르면 고려해야 하는 예외 처리 의존성은 다음의 두 가지이다.

- 서로 다른 의사 물리 블록으로 향하며 재사상 작업을 동반하는 요청 사이에 발생하는 의존성

- 재사상 작업을 동반하는 쓰기 요청과 동일 의사 물리 블록 내의 다른 페이지로 향하는 읽기 요청 사이에 발생하는 의존성

전자는 하나의 요청이 처리되는 시간 구간 안에 복수의 불량 블록이 발생하는 경우로써, 극히 드물게 발생하는 경우이기 때문에 성능보다는 구현 비용 절감에 중점을 맞추어 해결한다. 처리 중에 불량 블록이 발생한 요청을 살펴보면 해당 요청을 위한 재사상 작업만이 시스템 상태에 영향을 미치는 유효한 작업이며, 재사상 이전에 수행된 작업은 불량 블록 상에서 수행되었기 때문에 무효화되는 작업이다. 따라서 불량 블록이 발생한 서로 다른 두 요청에 대해서 재사상 작업을 순차적으로 처리한다면 해당 요청 간의 재사상에 의한 의존성이 해결된다. 이와 같이 재사상

작업 간의 중첩을 허용하지 않는 방식은 구현이 단순하다는 것 외에도 재사상을 위한 알고리즘이 변경에 상관없이 적용 가능하다는 장점이 있다. 정리한 실행 규칙은 다음과 같다.

실행 규칙4: 서로 다른 의사 물리 블록으로 향하며 재사상 작업을 동반하는 두 FTL요청에 대하여, 재사상 작업은 요청 순서에 따라 순차적으로 실행되어야 한다.

후자의 경우는 쓰기 요청 처리 중에 블록 재사상을 수행하였을 때 동일 의사 물리 블록으로의 읽기 요청들이 과거 또는 미래의 재사상 정보를 참조함으로써 발생하며, 재사상 정보 참조의 일관성 문제라고 볼 수 있다. 이와 같은 문제는 쓰기 순서 의존성을 해결하는 과정에서도 동일하게 발생하는 것이므로, 규칙 1, 2, 3를 쓰기 요청뿐만 아니라 읽기 요청까지 포괄하도록 확장함으로써 해결한다. 확장된 규칙들은 다음과 같다.

확장된 실행 규칙1: 동일 의사 물리 블록으로 향하는 두 FTL 요청에 대하여, 플래시 메모리 제어기로 연산 실행 요청을 순차적으로 전달한다.

확장된 실행 규칙2: 동일 의사 물리 블록으로 향하는 두 FTL 요청에 대하여, 선행 요청을 위한 재사상 작업이 시작된 후에 후행 요청이 도착하였다면, 후행 요청은 재사상 작업이 완료된 이후에만 처리가 시작되어야 한다.

확장된 실행 규칙3: 동일 의사 물리 블록으로 향하는 두 FTL 요청에 대하여, 선행 요청을 위한 재사상 작업이 시작하기 전에 후행 요청이 도착하여 처리가 시작되었다면, 후행 요청의 기존 작업은 무효화되고 재사상

작업이 완료된 이후에 재수행되어야 한다.

확장된 규칙들은 기존의 실행 제약의 적용 범위를 쓰기 의존성 관계에 있는 요청들에서 동일 의사 물리 블록으로 향하는 모든 요청들로 확장한 것이다. 확장된 규칙 1,2,3는 불량 블록이 발생하지 않는 대부분의 경우에 요청 처리 지연을 수반하지 않으므로 실행 제약 범위를 확장하였다고 하더라도 성능 저하가 거의 없을 것으로 기대된다. 유의할 점은 이와 같은 규칙을 엄격히 적용하지 않더라도 재사상에 의한 의존성이 해결될 수 있다는 것이다. 예를 들어 읽기 요청의 경우, 이전 버전의 재사상 정보를 참조하여 불량 블록 상에서 연산을 수행한다고 하더라도 불량 블록 내에서 반환한 데이터는 정확한 값을 가지고 있을 수 있다. 그러나 이러한 속성은 블록 재사상 알고리즘에 종속적인 것으로써, 재사상 알고리즘이 변경되거나 BMS의 기능이 확장된다면 정확성이 보장되기 어렵다. 예를 들어 [63]과 같이 불량 블록 내에 일부 정상 페이지를 재활용하도록 BMS의 기능을 확장한다면, 불량 블록을 특수 목적용 예비 블록으로 환원하기 위해 지우기 연산을 수행하므로 불량 블록에서의 읽기 작업 결과는 정확성을 보장할 수 없다. 즉, 확장된 규칙1,2,3는 성능 저하가 경미하다는 장점 이외에도 특정 블록 재사상 알고리즘에 종속되지 않고 의존성 문제를 해결하기 때문에, 개발자는 의존성 문제에 의한 제약 없이 BMS의 재사상 알고리즘을 변경하거나 기능을 확장할 수 있다.

3.3.4 논의

지금까지 병렬적인 요청 처리의 정확성을 위해 4가지 실행 규칙을 제시하였다. 규칙 1, 2, 3은 개별 의사 물리 블록의 관점에서 요청들이

순차적으로 처리된 것과 동일한 결과를 보장하기 위한 것이다. 순차적인 처리 과정에서는 모든 FTL 요청들에 대하여 주소 변환 작업 시 항상 최신의 사상 정보를 참조하게 되며, 주소 변환에 의해 발생한 물리 블록 연산들은 한 번에 하나씩 실행된다. 사상 정보 참조의 최신성은 병렬 처리 상황에서 불량 블록이 발생하더라도 규칙 2와 3을 통해 만족된다. 재사상 작업이 수행되면 사상 정보가 갱신되며 동일 의사 물리 블록에 대한 후행 요청들은 재사상 작업 완료 시점에 주소 변환 작업을 수행해야 최신의 사상 정보를 참조할 수 있다. 재사상 작업 이후에 발생한 요청들은 규칙 2에 의해서 재사상 완료 시점까지 요청 처리가 지연되므로 주소 변환 시 최신의 사상 정보를 참조하게 된다. 재사상 작업 이전에 발생한 요청들은 과거 버전의 사상 정보를 참조하여 플래시 메모리 연산을 실행할 수 있으나, 규칙 3에 의하여 실행된 연산의 결과는 무효화되고 재사상 완료 이후 시점에 사상 정보 참조 작업이 다시 수행된다. 따라서 어떠한 경우에도 최신의 사상 정보를 참조하여 주소 변환 작업이 수행된다. 주소 변환 결과로써 발생한 물리 블록 연산 요청은 규칙 1에 의하여 순차적으로 플래시 메모리 제어기로 전달되며, 플래시 제어기는 동일 물리 블록에 대한 연산을 한 번에 하나씩 실행하기 때문에 연산 실행 결과가 순차적인 경우와 동일함이 보장된다. 주목할 점은 과거 사상 정보를 참조하여 실행된 연산들의 효과가 불량 블록 상에 반영되므로 불량 블록의 내용은 순차적으로 요청을 처리하는 경우와 상이할 수 있다는 것이다. 그러나 재사상에 의해 불량 블록들은 시스템에서 더 이상 사용되지 않도록 배제되기 때문에 불량 블록의 내용은 정확성에 영향을 미치지 않는다.

규칙 4는 재사상 정보를 비롯한 메타 정보의 갱신 결과가 FTL 요청

들을 순차적으로 처리한 경우와 동일함을 보장하기 위한 것이다. 메타 정보는 재사상 작업이 수행되는 경우에만 갱신되며 순차 처리에서는 연산 간의 간섭이 존재하지 않기 때문에 갱신된 메타 정보는 항상 일관된 상태를 유지한다. 병렬 처리 상황의 경우를 고려해보면, 재사상 작업 진행 중에 다른 의사 물리 블록으로 향하는 FTL 요청들이 동시적으로 처리될 수 있으며 이 중 재사상 작업을 유발하는 요청들이 존재할 수 있다. 규칙 4를 준수한다면 모든 재사상 작업은 순차적으로 처리되기 때문에 재사상 작업 간의 간섭 문제는 발생하지 않는다. 재사상 작업을 유발하지 않은 요청들은 재사상 작업과 데이터 충돌 관계가 없으므로 재사상 작업에 영향을 주지 않는다. 따라서 다수의 FTL 요청이 병렬적으로 처리되더라도 재사상 작업의 결과는 순차 처리의 경우와 동일함이 보장되며 메타 정보의 일관성은 항상 유지된다.

정확성을 위한 실행 규칙 이외에도 순차적인 응답에 관한 규칙을 제시하였는데, 이는 기존의 순차적인 BMS와 인터페이스 측면의 동일성을 보장하기 위함이다. 응답 순서가 뒤바뀌더라도 FTL이 이를 인지하여 정확하게 처리하는 것이 가능하므로 순차적인 응답 규칙은 정확성을 위한 필수적인 조건은 아니다. 그러나 순차적인 응답으로 인해 FTL의 복잡도를 줄일 수 있는 장점이 있다. 호스트 요청 처리와 같이 FTL이 수행하는 작업은 보통 다수 개의 의사 물리 블록 연산 요청을 통해 처리되며 단일 작업을 구성하는 모든 연산이 완료되어야만 해당 FTL 작업이 완료됨이 보장된다. 예를 들어 호스트 요청에 대해서 해당 요청을 처리하기 위한 의사 물리 블록 연산들이 모두 완료되어야만 호스트 시스템에 완료 응답을 할 수 있다. 만약 비순차적으로 응답을 하는 BMS라면 작업을 구성하는 모든 연산에 대해서 완료 여부를 파악해야 되고 이를 위

해 별도의 자료 구조를 유지해야 하는 부담이 생긴다. BMS가 순차적으로 응답을 한다면 FTL에서는 작업을 구성하는 연산 중 마지막 연산의 완료만을 검사하면 되므로 FTL의 구현 부담이 크게 줄어들 것이다. 이와 같이 응답 순서에 관한 규칙은 순차적인 BMS와의 인터페이스 측면의 동일성을 확보함과 동시에 FTL의 구현 복잡도를 완화하는 역할을 한다.

제 4 장　병렬화된 불량 블록 관리 기법 설계

4.1 가정 및 설계 목표

본 장에서는 FTL 요청 간의 의존성 문제를 해결하면서, 플래시 메모리 및 제어기가 제공하는 병렬성을 효율적으로 활용하는 불량 블록 관리 기법(BMS)의 설계 및 구현 방안을 서술한다. 먼저 FTL과 같은 상위 계층 및 하위의 플래시 메모리 제어기에 대한 가정을 바탕으로 BMS의 역할을 규명하고 이를 통해 설계 목표를 설정한다. 기본 가정들을 정리하면 다음과 같다.

(1) 상위 계층은 자체적으로 데이터 의존성 문제를 해결한다.

3.2.1절에서 고찰한 바와 같이 FTL에서는 저장 장치의 일관성을 보장하기 위해 내부 작업 간의 동기화를 수행하며, 이에 따라 동시적으로 발생한 요청 간에 데이터 의존성이 성립하는 것은 생각하기 어렵다. 실례로 HIL 프레임워크로 통해 구현된 다양한 종류의 FTL에서는 내부 모듈 간의 동기화 규칙에 따라 자체적으로 데이터 의존성 문제를 해결한다. 따라서 BMS 상에서는 요청 간의 데이터 의존성이 없다고 가정하고 설계를 수행하도록 한다. 만약 이러한 가정이 성립하지 않는 FTL 구현이 존재한다면, BMS와 FTL 사이에 전처리기 (Pre-processor)를 추가하여 데이터 의존성 문제를 계층적으로 해결할 수 있다. 이러한 접근 방법은 BMS의 설계 복잡도를 크게 줄이는데 기여할 수 있으며, 데이터 의존성의 발생 확률이 매우 작다는 것을 생각하면 성능 저하 부담은 경미할 것으로 예상된다.

(2) 플래시 메모리 제어기는 확장성을 보장하는 패킷 기반의 인터페이스를 제공한다.

패킷 기반의 인터페이스는 NVM express[64]나 SCSI[65]와 같이 다수의 요청을 병렬적으로 처리하기 위한 저장 장치 인터페이스에서 널리 사용되며, 사용자는 패킷에 원하는 내용을 추가함으로써 인터페이스가 제공하는 기능을 손쉽게 확장할 수 있다. 이러한 장점으로 인해 패킷 기반의 인터페이스는 플래시 메모리 제어기 분야에도 적용되고 있다[6]. 패킷 인터페이스를 통해 BMS는 FTL 요청에 대한 정보를 패킷에 추가하여 플래시 제어기에 전달하게 되며, 불량 블록 발생 시 추가된 정보를 이용하여 의사 물리 블록의 내용을 복구한다. 따라서 처리 중인 (Outstanding) FTL 요청에 대한 정보를 BMS가 별도로 관리할 필요가 없어지며, BMS의 구현 복잡도가 크게 완화된다. 또한 추후 BMS의 성능을 개선하기 위한 기법을 도입하거나 기능을 추가하는 경우에 패킷의 내용을 확장하여 해당 작업을 손쉽게 수행할 수 있다.

(3) 플래시 메모리 제어기는 무순서 중첩 실행 방식에 기반 하여 동작한다. 단, 동일 물리 블록에 대한 연산들은 순차적으로 수행된다.

대부분의 플래시 메모리 제어기는 다수 개의 플래시 메모리 칩을 최대한 병렬적으로 활용하기 위한 연산 스케줄링 작업을 수행한다. 구체적인 스케줄링 정책과 구현 방식은 플래시 제어기 별로 상이하며, BMS 설계 시 특정 스케줄링 정책을 가정한다면 BMS의 범용성을 확보할 수 없을 것이다. 무순서 중첩 실행 방식[6]에서는 플래시 메모리 연산들이 요청된 순서와 상관없이 임의의 순서로 중첩되어 실행된다. 따라서 무순서 중첩 실행 방식에서도 정확하게 동작하는 BMS는 실행 순서 제약이 부과된 다른 스케줄링 방식을 사용하는 제어기에서도 정확하게 동작할 것이다. 단, BMS가 의존성 문제를 해결하기 위한 최소한의 요구 사항으로서 동일 물리 블록에 대한 순차 실행은 보장되어야 한다. 이와 같은 제약이 지켜지지 않는다면 BMS에

서 효율적인 방식으로 의존성문제를 해결하는 것이 어려워진다. 상용 플래시 메모리 제어기들은 칩 또는 LUN 단위의 연산 스케줄링 기법을 채택하고 있는데 이는 서로 다른 칩 (또는 LUN)으로 향하는 연산들은 중첩하여 실행하지만 동일 칩 (또는 LUN)에 대해서는 순차적으로 연산을 수행하는 방식이다. 이러한 스케줄링 기법은 상기 가정에 비해 실행 순서 제약이 강화된 것이며, 상기 가정에 기반하여 설계된 BMS는 상용 플래시 제어기에서도 적용될 수 있음을 의미한다.

(4) 플래시 메모리 제어기는 요청된 순서와 동일한 순서로 연산 완료 응답을 전달한다.

기존의 순차 실행에 기반한 BMS와 인터페이스 측면의 동일성을 확보하기 위해 제안하는 BMS에서는 요청된 순서대로 FTL에 처리 완료 응답을 보낸다. 이를 구현하기 위해서는 완료 응답을 임시 저장하고 재정렬(Reordering)하는 기능이 필요한데, 이는 구현 복잡도를 심화시키고 하드웨어 자원 소모량을 증가시킨다. 무순서 플래시 메모리 제어기[6]는 재정렬 버퍼를 내장하여 연산 실행 결과를 반환한다. 이와 같은 플래시 제어기의 특성을 활용하면 BMS는 불량 블록이 발생하지 않은 대부분의 경우에 응답 재정렬을 할 필요가 없으므로 효율적인 구현이 가능해진다. FTL의 특성에 따라 순차적인 응답 기능은 BMS에서 제외될 수도 있으며, 이를 감안하면 해당 기능을 위한 구현 복잡도를 최소화하는 것이 바람직하다.

상기의 가정들을 바탕으로 BMS의 역할 및 설계 목표를 정리하면 다음과 같다.

(1) 데이터 의존성 문제는 FTL이나 BMS 앞단의 전처리기가 해결하며, BMS는 쓰기 순서 및 재사상에 의한 의존성 문제 해결을 담당한다. 데이터

의존성 해결 메커니즘을 BMS와 독립시킴으로써 BMS가 해결해야 하는 문제의 범위는 크게 감소되며 다양한 성능 개선 기법을 보다 수월하게 적용할 수 있게 된다. 의존성 해결 메커니즘을 통해 BMS 처리 결과의 정확성이 보장되어야 하며, 이는 병렬 처리의 결과가 순차 실행 결과와 동일해야 함을 의미한다.

(2) 병렬적으로 발생한 상위 계층 요청에 대해서, 지연 시간을 최소화하여 플래시 제어기로 전달함으로써 플래시 메모리 제어기에 충분한 양의 작업 부하 (Workload)를 공급한다. 연산을 스케줄링하여 플래시 메모리 칩을 최대한 병렬적으로 활용하는 것은 무순서 중첩 실행 방식의 플래시 메모리 제어기가 전담하며, 스케줄링 효율을 높이기 위해 작업 부하를 생성하는 것은 BMS가 담당한다. 이와 같은 목표를 달성하기 위하여 BMS로 요청 전달의 지연이 최소화되어야 하며, 자주 수행되는 기능은 하드웨어로 구현하여 처리 속도를 극대화한다.

(3) 하드웨어로 구현된 기능은 저장 장치 비용 절감을 위해 적은 양의 자원만으로 동작할 수 있어야 하며, 다양한 응용에 적용될 수 있도록 규모 확장성 (Scalability)를 보장해야 한다.

4.2 전체 구조

본 절에서는 다수의 FTL 요청을 병렬적으로 처리하는 BMS (이하 병렬화된 BMS)의 구조를 설명한다. 설계 시 고려된 사항을 명확히 하고 설계 과정의 이해를 돕기 위해 귀납적인 방법으로 전체 구조 설계를 설명한다. 먼저 순차적으로 동작하는 BMS의 구조를 통해 핵심적인 작업을 설명하고,

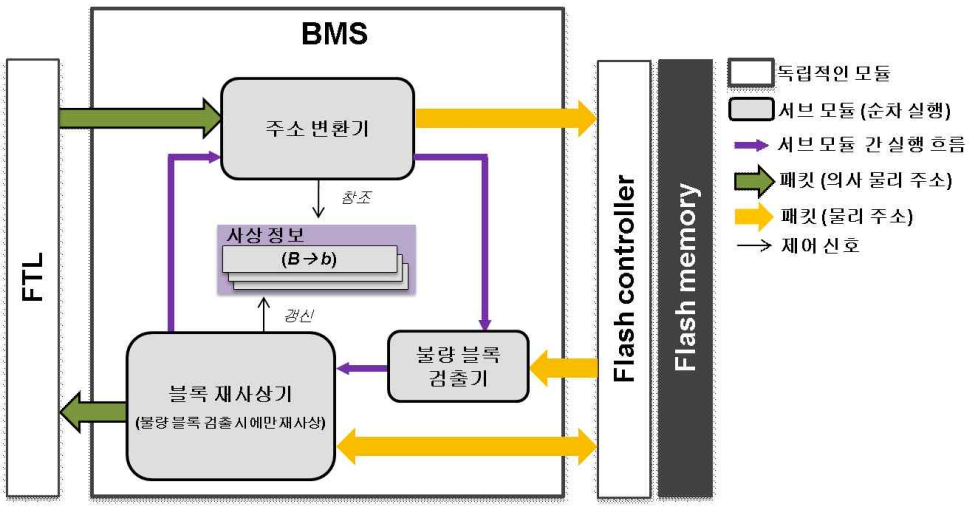


그림 24 순차적으로 동작하는 BMS 구조

작업 수준 병렬화를 통해 기본적인 병렬 설계 구조를 도출한다. 또한 의존성 해결을 위한 실행 규칙들과 인터페이스 동일성 보장을 위한 응답 규칙을 만족시키는 모듈 및 메커니즘을 추가하여 최종 구조를 확립한다.

4.2.1 순차적으로 동작하는 BMS의 구조

그림 24는 순차적으로 동작하는 BMS의 내부 구조를 보여 준다. 순차적인 BMS는 한 번에 하나의 상위 계층 요청만을 처리하기 때문에 내부 작업들이 직렬화되어 수행된다. 상위 계층으로부터 요청이 들어오면 주소 변환기는 재사상 정보를 참조하여 의사 물리 블록 주소를 물리 주소로 변경한다. 재사상되지 않은 의사 물리 블록의 경우, 재사상 정보 참조 결과가 무의미한 (Null) 물리 블록 주소를 얻게 되는데, 이 경우 주소 변환기는 기본 사상을 통하여 물리 주소를 얻게 된다. 주소가 변환된 요청은 플래시 메모리 제어기로 전달되며, BMS는 요청한 연산에 대한 완료 응답이 플래시 제어기로부터 도착할 때까지 기다린다. 완료 응답

결과로 불량 블록의 발생이 감지되면 의사 물리 블록 재사상 작업을 수행하고 그렇지 않으면 상위 계층으로 완료 응답을 전달한다. 재사상 작업에서는 불량 블록을 정상 블록으로 교체하고 연산이 성공적으로 완료된 것과 동일한 상태로 의사 물리 블록을 복구한다. 복구 과정 중에 다수의 플래시 메모리 페이지와 메타 데이터의 내용이 갱신될 수 있으나, 다른 의사 물리 블록에 사상된 물리 블록의 내용과 관련된 사상 정보는 변경되지 않는다. 재사상 작업이 완료되면 해당 의사 물리 블록에 대한 요청이 성공적으로 완료된 상태가 되므로, FTL로 완료 응답을 전달한다. 이러한 전체 과정이 수행 완료된 이후에 새로운 FTL 요청이 처리될 수 있다.

4.2.2 병렬화된 BMS의 기본 구조

순차적인 BMS의 동작 과정을 살펴보면 플래시 메모리 및 제어기로 입출력 요청을 하고 기다리는데 대부분의 시간이 소비될 것이라고 예측할 수 있다. 플래시 메모리의 연산 지연 시간은 수십 us에서 수 ms 정도이며, 저장 장치 시스템의 다른 구성 요소에 비해 처리 속도가 매우 느리기 때문이다. 더욱이 BMS의 대기 시간은 불량 블록이 발생하는 경우 심각하게 길어지는데, 재사상 작업에서 수십 ~ 수백 개의 플래시 메모리 연산이 발생하기 때문이다. 이러한 대기 시간을 활용하여 다른 유용한 작업을 수행하도록 하는 것이 작업 수준 병렬화의 기본적인 목적이다.

그림 25는 설명한 세 가지 작업이 병렬적으로 수행될 수 있도록 변경한 BMS의 구조이다. 주소 변환기는 FTL 요청을 변환하여 플래시 제어기로 전달하며, 플래시 메모리 연산의 완료 여부와 상관없이 새로운

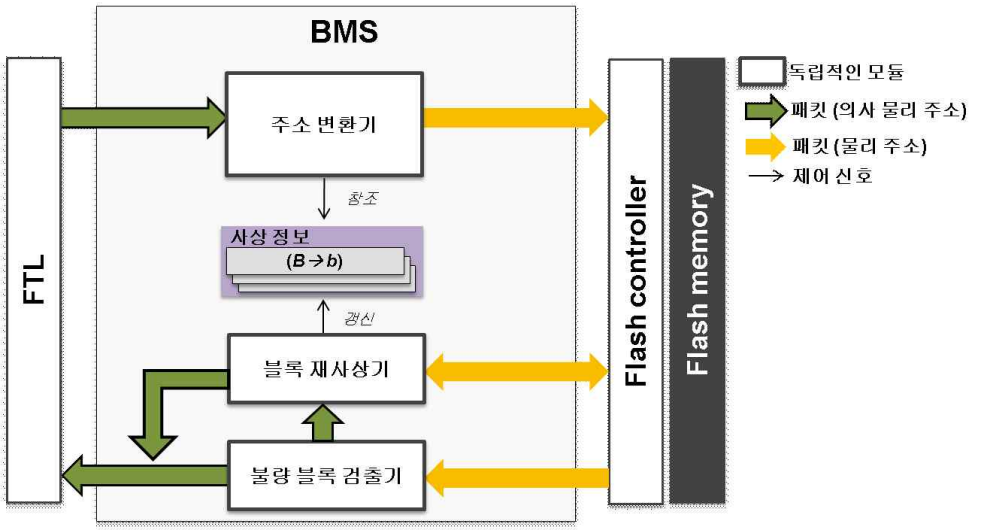


그림 25 병렬화된 BMS 기본 구조 (정확성 보장 못함)

FTL 요청에 대한 변환 및 전달 작업을 수행할 수 있다. 따라서 FTL 요청이 짧은 시간 안에 집약적으로 발생하였을 때, 요청 전송 모듈은 연속적으로 해당 요청을 플래시 제어기에 공급하게 되며, 플래시 제어기는 충분한 양의 작업 부하를 확보하였기 때문에 효율적으로 연산 스케줄링을 하는 것이 가능하다. 플래시 제어기의 완료 응답은 불량 블록 검출기로 전달되는데, 불량 블록이 발생하지 않은 경우에는 FTL로 응답을 전달하며 이후 새로운 완료 응답에 대한 처리를 수행한다. 불량 블록이 발생한 경우에는 완료 응답을 블록 재사상기로 전달하는데, 재사상 작업이 끝나는 것을 기다리지 않고 플래시 제어기로부터 도착한 다음 응답을 처리하기 때문에 응답 전송에 대한 처리 지연은 발생하지 않는다. 재사상기는 불량 블록 검출기로부터 전달 받은 응답에 대해서 재사상 작업을 수행한다. 재사상 작업 중에 다수의 플래시 연산으로 인한 입출력 대기 시간이 반복적으로 발생하기 때문에 재사상 작업 동안에 CPU와 같은

자원을 독점적으로 사용하는 것은 바람직하지 않다. 재사상기는 대기 시간이 발생할 때마다 사용 중인 자원을 점유 해제 (Release)함으로써 BMS의 전체 작업의 효율성을 높일 수 있다. 재사상 작업 결과로서 성공적인 FTL 요청 처리 결과가 FTL로 전달된다.

그림 25와 같은 구조에서는 다수의 FTL 요청이 병렬적으로 실행되며, 재사상 작업으로 인해 플래시 메모리 제어기 상에서 병렬적으로 실행되는 연산의 수는 FTL 요청보다 많을 수 있다. 이와 같은 병렬 처리 상황에서는 플래시 메모리 연산에 대한 완료 응답이 도착했을 때, 해당 연산을 발생시킨 작업이나 FTL 요청을 식별하기 위한 정보가 필요하다. 4.1 절에서 가정한 바와 같이 플래시 메모리 제어기는 확장 가능한 패킷 인터페이스를 가지며, BMS는 패킷 내 여분의 필드를 예약해 둠으로써 패킷 별로 필요한 정보를 추가할 수 있다. 플래시 연산 패킷에는 FTL 요청 식별 정보 (Identification; ID), 작업 식별 정보 (정상 작업 또는 재사상 작업), 의사 물리 블록 주소가 포함될 수 있다. 식별 정보들은 연산과 관련된 FTL 요청 및 작업들을 파악하기 위한 것이며, 의사 물리 블록 주소는 재사상 작업에 필요한 정보이다. 이와 같은 정보들은 요청 전송 모듈에서 생성하여 플래시 연산 패킷에 추가된다.

그림 25의 병렬 처리 구조는 대기 시간에 유용한 작업을 수행하도록 함으로서 BMS의 작업 효율성을 높이는 것을 그 목적으로 한다. 그러나 병렬 처리 과정에서 발생하는 연산 간의 의존성 문제를 고려하지 않으므로, 동작의 정확성을 보장할 수 없는 한계가 있다. 또한 플래시 메모리 연산이나 재사상 작업이 완료된 순서에 따라 FTL에 대한 응답을 수행하기 때문에 순차적인 BMS와 동일한 인터페이스를 제공하지 못한다. 이는 병렬화된 BMS의 호환성 및 범용성 문제를 초래하는데, 순차적인

BMS를 가정하고 설계된 FTL의 경우 병렬화된 BMS를 적용할 수 없기 때문이다. 이와 같은 문제를 해결하기 위해 3장에서 확립한 실행 규칙 및 응답 규칙을 만족하는 모듈 및 메커니즘이 BMS에 도입되어야 한다.

4.2.3 정확성을 보장하는 병렬화된 BMS 구조

의존성 문제를 해결하기 위해 정리된 규칙들은 다음과 같다.

실행 규칙1: 동일 의사 물리 블록으로 향하는 두 FTL 요청에 대하여, 플래시 메모리 제어기로 연산 실행 요청을 순차적으로 전달한다.

실행 규칙2: 동일 의사 물리 블록으로 향하는 두 FTL 요청에 대하여, 선행 요청을 위한 재사상 작업이 시작된 후에 후행 요청이 도착하였다면, 후행 요청은 재사상 작업이 완료된 이후에만 처리가 시작되어야 한다.

실행 규칙3: 동일 의사 물리 블록으로 향하는 두 FTL 요청에 대하여, 선행 요청을 위한 재사상 작업이 시작하기 전에 후행 요청이 도착하여 처리가 시작되었다면, 후행 요청의 기존 작업은 무효화되고 재사상 작업이 완료된 이후에 재수행되어야 한다.

실행 규칙4: 서로 다른 의사 물리 블록으로 향하며 재사상 작업을 동반하는 두 FTL 요청에 대하여, 재사상 작업은 요청 순서에 따라 순차적으로 실행되어야 한다.

또한 순차적인 BMS와 동일한 인터페이스를 FTL에 제공하기 위한 규칙은 다음과 같다.

응답 규칙: 모든 요청 처리에 대한 완료 응답은 FTL에 순차적으로 전달

한다.

이제 상기 정리한 규칙을 만족시키기 위해 필요한 자료 구조 및 기능을 고찰하도록 한다. 그림 25의 구조에서는 불량 블록이 발생하지 않는 한 상기 규칙들이 자연스럽게 만족된다. FTL 요청은 순차적으로 주소 변환되어 플래시 제어기에 전달하기 때문에 실행 규칙1을 만족하며, 재사상이 수행되지 않기 때문에 실행 규칙 2, 3, 4는 고려될 필요가 없다. 플래시 제어기에서 요청 순서와 동일한 순서로 응답함을 가정하므로, 응답 전송 모듈에서 별다른 응답 재정렬 기능을 지원하지 않더라도 응답 규칙도 자연스럽게 만족된다. 따라서 불량 블록이 발생하는 경우에 집중하여 고찰 과정을 심화시키도록 한다.

불량 블록으로 인해 의사 물리 블록이 재사상되는 경우, 규칙2를 만족하기 위하여 추후 동일 의사 물리 블록에 대한 요청들은 재사상이 완료될 때까지 처리 작업이 지연되어야 한다. 그러나 다른 의사 물리 블록으로의 요청들은 지연 없이 곧바로 처리될 수 있어야 하는데, 플래시 제어기에 충분한 작업 부하를 공급하는 것이 병렬화된 BMS의 기본적인 목표이기 때문이다. 이와 같이 선택적으로 요청에 대한 처리 적업을 지연시키기 위해서는 의사 물리 블록의 재사상 진행 여부를 사전에 알 수 있어야 한다. 또한 지연된 요청들을 별도의 메모리에 유지해야 하는데 이는 재사상 완료 후에 해당 요청들을 수행해야만 하기 때문이다.

재사상 수행 여부가 결정되는 시점은 쓰기 및 지우기 연산 오류로 인해 불량 블록이 감지될 때이며, 이는 상위 계층 요청이 도착한 시점에서부터 (BMS 및 상위 계층의 관점에서) 상당히 긴 시간이 지난 이후 시점이다. 따라서 불량 블록이 감지되기 전에 동일 의사 물리 블록으로 다

수의 요청이 도착하고 플래시 제어기로 연산 실행 요청이 전송될 수 있다. 해당 요청들은 재사상 작업 이전 버전의 재사상 정보를 참조하게 되므로 불량 블록에서 연산이 실행될 것이며, 이를 해결하기 위한 규칙 3을 만족시키려면 불량 블록에서 실행된 요청에 대한 작업 무효화 기능이 필요하다. 작업 무효화 여부는 플래시 제어기로부터 응답을 전달 받는 시점에 판단할 수 있을 것이다. 응답 패킷에서 상위 계층 요청을 추출하여 대상 의사 물리 블록의 재사상 진행 여부를 파악하고, 재사상 진행 중이면 해당 요청은 무효화되어 재사상 작업 완료 이후 처음부터 처리 과정이 재수행된다. 이와 같이 무효화된 요청들은 처리 지연된 요청과 마찬가지로 별도의 메모리에 유지되어야 한다. 주의할 점은 재사상 완료 이후 시점에서 해당 의사 물리 블록에 대한 요청들을 처리할 때, 규칙 1에 따라 순차적으로 처리되어야 한다는 것이다. 처리 대상 요청들은 세 가지 그룹으로 분류되는데, 무효화된 요청들, 지연된 요청들, 재사상 완료 후 새롭게 발생한 요청들로 분류된다. 규칙 1을 만족시키기 위해서는 무효화된 요청들이 가장 먼저 순차적으로 재수행되어야 하며, 그 후 지연된 요청들이 순차적으로 수행되어야 한다. 이와 같은 요청들에 대한 처리가 순차적으로 시작된 후에만 새롭게 발생한 요청들이 처리될 수 있다.

이와 같이 규칙 2와 3을 만족시키는 과정에서 서로 다른 의사 물리 블록에 대한 요청들은 순서가 뒤바뀌어 플래시 제어기에 전달될 수 있으며, 이는 플래시 제어기의 순차적인 응답 순서가 규칙 5에서 규정한 FTL에 대한 순차적인 응답 순서와 상이할 수 있음을 의미한다. 따라서 순서 뒤바뀜을 판단하여 상위 계층에 대한 응답 순서를 재정렬하는 기능이 필요할 것이다. 재정렬을 위해서는 FTL 응답들을 임시로 유지하기

위한 메모리가 필요한데, 순서 뒤바뀜은 재사상 작업으로 인해서만 발생한다는 사실을 감안하면 대부분의 경우 재정렬 작업은 필요가 없을 것이며 재정렬을 위한 메모리 사용 빈도도 극히 낮을 것이다.

병렬화된 BMS에서는 다수의 요청들이 동시에 수행되기 때문에 불량 블록이 동시 다발적으로 발생하는 것도 가능하다. 이와 같은 상황을 고려하여 도입된 규칙 4를 만족시키기 위해서, 재사상 작업을 FTL 요청 순서에 따라 한 번에 하나씩 실행하는 방법을 생각할 수 있다. 이러한 방법은 재사상 작업의 직렬화 (Serialization)를 유발하지만, 복수의 불량 블록이 동시에 발생하거나 재사상 작업 중에 추가의 불량 블록이 발생하는 것은 극히 드문 경우이기 때문에 직렬화로 인한 성능 저하는 거의 없을 것이다. 규칙 2와 3을 만족시키는 과정에서 서로 다른 요청들은 순서가 뒤바뀌어서 플래시 제어기로 전달될 수 있으므로, 불량 블록이 감지되는 순서는 FTL 요청 순서와 다를 수 있다. 이와 같은 상황에서 재사상 작업 간의 순서를 맞추기 위해서는 앞서 설명한 응답 순서 재정렬 기능을 활용할 수 있다. 재사상 작업이 필요한 특정 요청에 대해서, 선행 요청들이 모두 응답 완료되었다는 것이 확인되면 선행 요청들을 위한 재사상 작업도 모두 완료되었음이 보장된다. 따라서 응답 순서 재정렬 모듈이 제공하는 정보를 참조하여 재사상 작업의 순차적인 처리를 보장할 수 있다.

고찰한 내용을 바탕으로 정확성을 보장하는 병렬화된 BMS의 구조는 그림 26과 같다. FTL 요청이 도착하면 요청 필터에 의해 재사상 중인 블록으로 가는 요청들의 처리가 지연된다. 요청 필터는 재사상 진행 블록 큐에 질의하여 FTL 요청이 접근하는 의사 물리 블록이 재사상 중인지를 판단한다. 재사상 중이면 해당 요청은 대기 요청 큐로 옮겨져서 재

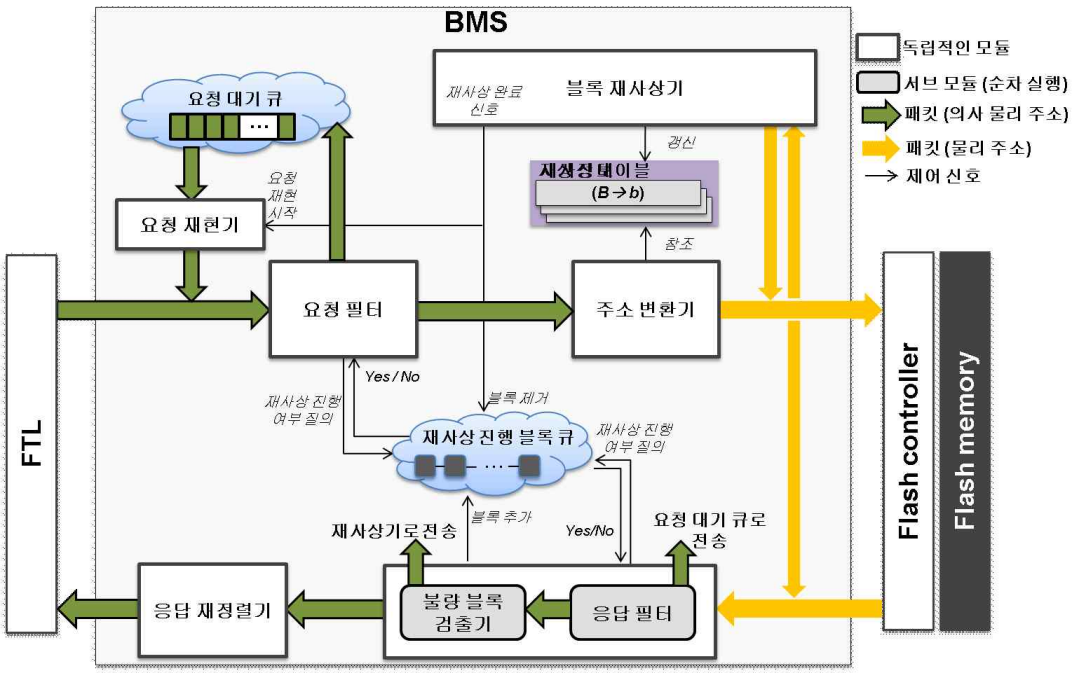


그림 26 정확성 보장을 위한 병렬화된 BMS 구조

사상이 완료될 때까지 처리가 지연되며, 그 외의 경우에는 정상적인 처리 (주소 변환 및 플래시 제어기로 요청 전송)가 진행된다.

플래시 제어기로부터 도착한 응답은 응답 필터 상에서 두 단계를 통해 걸러진다. 먼저 응답으로부터 FTL 요청을 복원하고 재사상 작업 상태 관리자에 질의하여 대상 의사 물리 블록이 재사상 중인지를 판단한다. 재사상 중이면 불량 블록 상에서 연산이 실행된 요청이므로, 진행된 작업 결과를 무효화하고 재사상 완료 이후에 재수행하기 위해 해당 요청을 대기 요청 큐로 옮긴다. 정상 블록에서 수행된 요청에 대해서는 지우기 및 쓰기 오류 여부를 검사하며, 검사 결과 불량 블록이 발생한 경우, 재사상 작업 상태 관리자로 해당 요청을 전달한다. 불량 블록이 발생하

지 않았다면 응답 재정렬기로 요청을 전달한다. 응답 재정렬기는 가장 최근에 응답된 요청의 식별 정보를 유지하며 새로운 요청이 도착하면 식별 정보를 비교하여 해당 요청이 응답될 차례인지 검사한다. 응답 가능한 차례가 아니라면 내부 버퍼 (재정렬 버퍼)에 요청을 저장한다. 응답 가능하다면 FTL로 응답을 전달하고 최근 식별 정보를 갱신한다. 또한 재정렬 버퍼에 응답 지연된 요청이 있는 경우, 식별 정보 비교 작업을 통해 추가의 응답 작업을 수행하는데, 이러한 작업은 재정렬 버퍼에 응답 가능한 요청이 없어질 때까지 반복된다. 플래시 제어기가 순차적으로 연산 응답을 발생시킨다는 사실을 고려하면, 불량 블록이 발생하지 않는 대부분의 경우 재정렬이 필요하지 않으므로, 재정렬 작업으로 인한 버퍼 사용량 및 응답 지연은 무시할 수 있는 수준일 것이다.

불량 블록 발생이 감지되어 재사상 작업이 필요한 요청들은 블록 재사상기로 전달되며, 해당 요청들의 의사 물리 블록 주소들은 재사상 진행 블록 큐에 기록된다. 재사상기는 복수의 재사상 작업들이 있는 경우 FTL 요청 순서에 따라 순차적으로 재사상 작업을 수행한다. 재사상 작업이 완료되면 해당 요청은 재정렬 버퍼를 거쳐 FTL로 응답되고, 대응되는 의사 물리 블록의 주소는 재사상 진행 블록 큐에서 제거된다. 또한 해당 재사상 작업으로 인해 무효화되었거나 지연된 요청들을 대기 큐에서 꺼내서 순차적으로 실행하는데 이는 요청 재현기가 담당한다. 유의할 점은 재사상 작업 완료 시점부터 요청에 대한 재현이 완료되는 시점까지 새로운 요청에 대한 처리가 금지된다는 것이다. 이는 재현되는 요청이 새롭게 들어오는 요청보다 처리 순서가 선행하기 때문이다. 요청 재현 작업은 단순히 대기 큐로부터 요청 필터로 요청들을 전달하는 작업으로서 복잡도가 작으며 재현하는 요청의 수는 최악의 경우에도 블록 내의

페이지 개수보다 작을 것이기 때문에, 새로운 요청 처리가 금지되는 시간으로 인한 성능 저하 부담은 경미할 것이다.

4.3 하드웨어 기반 최적화 설계

기존의 BMS는 소프트웨어로 구현되며 FTL과 같은 상위 계층과 동일한 프로세서 상에서 구동된다. 병렬화된 BMS도 이와 동일한 방식으로 구현될 수 있으나 다음과 같은 문제점이 있다.

(1) FTL과 프로세서 및 메모리 자원을 공유함으로서 FTL의 성능을 저하시킬 수 있다.

FTL은 플래시 메모리의 단점을 극복하고 효율적인 저장 장치 서비스를 제공하기 위해 다수의 복잡한 작업을 수행한다. 예를 들어, 논리 섹터와 플래시 메모리 페이지 간의 사상, 쓰레기 수집, 마모도 평준화, 버퍼 및 캐시 관리, 전원 오류 복구와 같은 다양한 작업을 수행하며, 이로 인해 전체 저장 장치 시스템에서 병목 현상이 될 가능성이 높다. BMS가 FTL과 자원을 공유하는 경우, 자원 점유율 감소로 인해 FTL의 성능이 저하될 것이며, 이는 FTL 병목 현상 문제를 심화시킬 것이다.

(2) 소프트웨어 작업에 의한 요청 전달 지연으로 인해 플래시 메모리 제어기에 대한 활용도가 감소할 수 있다.

플래시 메모리 제어기가 연산 스케줄링을 효율적으로 수행하기 위해서는, 짧은 시간 안에 다량의 플래시 메모리 연산 요청을 플래시 제어기로 전달해야 한다. 특히 복수의 연산 요청을 묶어서 다중 플레인 연산이나 캐시 연산으로 병렬 처리하는 플래시 제어기의 경우, 연산 요청이 연속적으로 (Burst) 발생하는 것이 중요하다. 소프트웨어로 구현된 BMS는

블록 주소 변환이나 의존성 해결 과정에서 하드웨어에 비해 훨씬 긴 지연 시간이 발생하므로, 빠른 속도로 플래시 연산 요청을 발생시키기 어렵다.

(3) BMS 내부의 작업의 병렬화는 프로세서 개수에 의해 제한된다.

그림 26과 같이 BMS는 다수의 모듈로 구성되며, 최소한의 동기화를 제외하고는 모듈들은 동시에 동작할 수 있다. 그러나 BMS 모듈들이 하나 또는 적은 수의 프로세서 상에서 구동된다면, 물리적으로 동시에 수행되는 모듈들의 개수는 프로세서 개수에 의해 제한될 것이다. 일반적으로 플래시 메모리 저장 장치에서는 비용 문제로 인해 많은 수의 프로세서를 사용할 수 없기 때문에 이와 같은 병렬성의 제한은 플래시 제어기를 효과적으로 활용하지 못하는 결과를 초래할 수 있다.

BMS를 특화된 하드웨어로 구현한다면 상기 제시된 대부분의 문제를 해결할 수 있다. BMS가 별도의 하드웨어로 독립되기 때문에 상위 계층은 프로세서 자원을 독점하여 사용할 수 있으므로 BMS로 인한 상위 계층 성능 저하 문제가 크게 개선된다. 또한 소프트웨어에 비해 동일 작업을 빠른 시간 안에 처리할 수 있으므로 플래시 제어기에 대한 요청 전달 시간이 크게 감소하고, BMS 내부 모듈이 독립적으로 동작하도록 구현하여 병렬성을 충분히 확보할 수 있다.

그러나 하드웨어 구현 시 다음과 같은 문제점이 있으며, 이어지는 절에서는 각각의 해결 방안을 제시하도록 한다.

(1) BMS는 재사상 모듈과 같이 복잡한 작업을 수행하기 때문에 전체를 하드웨어로 구현하는 경우 저장 장치 제품의 비용이 크게 증가할

수 있다. 플래시 메모리 저장 장치에서 비용은 제품의 경쟁력을 좌우는 중요한 요소이므로, 높은 구현 비용은 하드웨어 BMS의 범용성을 제한한다. 또한 기능 확장이나 성능 개선 등의 목적을 위해 BMS의 설계가 변경되는 경우, BMS 하드웨어를 처음부터 다시 구현해야 하는 문제가 있다. 이러한 문제점을 해결하기 위해 하드웨어와 소프트웨어가 혼합된 구현 방식을 도입할 수 있는데, 상세한 내용은 4.3.1절에서 설명한다.

(2) BMS가 사용하는 자료 구조는 주메모리 상에서 유지되며, 재사상 테이블이나 재사상 작업 상태와 같은 일부 정보는 매 요청 처리 시마다 접근되는 자료 구조이다. 따라서 FTL과 BMS는 주메모리를 공유하게 되며, 이는 메모리 접근의 병목으로 인해 상위 계층과 BMS의 성능을 크게 저해할 수 있다. 또한 BMS의 주소 변환 과정에서 재사상 테이블을 탐색하는 오버헤드 (Overhead)가 있는데 이는 BMS의 내부적인 병목 현상을 일으키는 부분으로 BMS의 처리 속도를 제한하게 된다. 따라서 주메모리 공유 문제를 해결하기 위한 효율적인 정보 관리 방식이 필요하며, 특히 재사상 정보 참조를 가속화할 수 있는 기법이 필수적으로 요구된다. 4.3.2절에서 관련된 내용을 상세히 설명한다.

4.3.1 하드웨어 지원 설계

하드웨어 비용 문제를 해결하고 설계 변경 사항을 쉽게 구현에 반영하도록 설계 유연성 (Flexibility)를 확보하기 위해 하드웨어와 소프트웨어가 혼합된 형태로 BMS를 구현한다. 이 때 중요한 고려 사항은 전체가 하드웨어로 구현된 경우와 비슷한 수준의 성능을 제공하는 것이다. 이를 위해 요청 처리 시마다 동작하는 모듈은 되도록 하드웨어로 구현하고 예외 상황 발생 시에만 동작하는 모듈은 소프트웨어로 구현하는 방법

을 취할 수 있다. 그림 26의 BMS 구조에서는 동작 빈도에 따라 모듈들을 쉽게 분류할 수 있다.

빈번하게 동작하는 모듈 및 접근되는 자료 구조는 다음과 같다.

모듈	동작 시점
요청 필터	FTL 요청이 도착할 때마다 동작
주소 변환기	FTL 요청이 플래시 제어기에 전달될 때마다 동작 수행
응답 필터	플래시 제어기로부터 응답이 도착할 때마다 동작
불량 블록 검출기	플래지 제어기 응답이 응답 필터를 통과할 때마다 동작
응답 재정렬기	정상 완료된 응답이 도착할 때마다 동작.

자료 구조	접근 시점
재사상 테이블	블록 주소 변환이 일어날 때마다 참조. 재사상 작업이 일어날 때마다 내용 갱신
재사상 진행 블록 큐	요청 필터 및 응답 필터가 동작할 때마다 참조됨, 재사상 작업이 시작하거나 완료될 때 갱신됨. (재사상 작업은 드물게 발생하므로 대부분의 경우 큐는 비어 있음)

예외 상황 시에만 동작하는 모듈과 접근되는 자료 구조는 다음과 같다.

모듈	동작 시점
블록 재사상기	불량 블록이 검출되는 경우에 동작
요청 재현기	재사상 완료 후 요청 대기 큐에 연관되는 요청이 있는 경우에 동작

자료 구조	접근 시점
요청 대기 큐	재사상 진행 중인 의사 물리 블록으로 요청이 들어오거나, 불량 블록 상에서 연산이 수행된 경우에만 접근됨
재정렬 버퍼	블록 재사상으로 인해 FTL 요청 수행 순서가 바뀌는 경우에만 접근됨. (대부분의 경우 재정렬 버퍼는 비어 있음)

빈번하게 동작하는 모듈들은 단순한 기능만을 필요로 하므로 하드웨어로 구현된다. BMS의 블록 관리 방식과 밀접히 연관되는 부분은 블록 재사상 작업이며, 해당 모듈들은 특정 블록 재사상 알고리즘에 종속되지 않기 때문에 BMS가 변경되더라도 해당 모듈들은 그대로 재사용될 수 있을 것이다. BMS의 기능이 확장되는 경우에 해당 모듈을 변경하는 것보다는 추가의 기능을 담당하는 모듈을 도입하는 방식을 취한다면 이러한 재사용성은 유지될 수 있다.

빈번하게 접근되는 자료 구조 중 재사상 테이블은 유지되는 정보의 양이 많기 때문에 주메모리 상에서 관리되며, 이는 상위 계층과 주메모리에 대한 자원 충돌 문제를 일으키게 된다. 또한 해시 테이블이나 트리(Tree)와 같이 다양한 형태로 관리될 수 있으며, 자원이나 성능에 대한 요구 사항에 따라 관리 방식이 변경될 수 있다. 따라서 해당 자료 구조 전체를 하드웨어를 통해 관리하는 것은 현실적으로 불가능하다. 이에 대한 해결책으로 전체 재사상 테이블은 소프트웨어로 관리하지만 자주 접근되는 일부 내용은 하드웨어로 관리되는 캐시에 담아서 FTL과 주메모리 공유로 인한 성능 저하를 줄이도록 한다. 또한 캐시 관리 방식을 전체 테이블 관리 방식과 독립적이 되게 함으로써 설계 유연성을 확보하도록 한다. 상세한 내용은 4.3.2절에서 다루도록 한다.

재사상 진행 블록 큐는 재사상 진행 중인 블록들의 리스트로서 재사상 작업이 매우 드물게 발생한다는 사실을 고려하면 대부분의 경우에 리스트는 비어 있는 상태일 것으로 예상할 수 있다. 또한 복수의 요청에 대한 재사상 작업이 동시에 필요한 경우는 극히 드물기 때문에 리스트에 속하는 블록의 개수는 대부분의 경우 레지스터 파일 (Register file)이나 소량의 캐시에 담을 수 있을만한 정도일 것이다. 따라서 리스트를 위해 소량의 고속 메모리를 할당하고 하드웨어로 관리하는 방식을 택할 수 있다. 재사상되는 블록이 고속 메모리에 담을 수 없을 만큼 많이 발생하는 극단적인 경우에는 소프트웨어가 개입하여 주메모리 상에서 블록들을 관리할 수 있는데 이와 같은 경우는 실제로 발생 확률이 거의 없을 것이다. 상세한 내용은 4.3.2절에서 다루도록 한다.

예외 상황 시에만 동작하는 모듈은 성능에 영향을 거의 주지 않기 때문에 소프트웨어로 구현한다. 특히 재사상기는 동작 과정이 복잡하고

BMS의 설계 유연성과 밀접히 연관되므로 소프트웨어로 구현하는 것이 바람직하다. 요청 재현기는 단순한 기능만을 수행하므로 하드웨어로 구현하는 것도 가능하나, 성능 개선을 기대하기 어렵기 때문에 하드웨어 비용을 낮추기 위해 소프트웨어로 구현하도록 한다.

요청 대기 큐, 재정렬 버퍼, 기타 BMS 메타 데이터는 접근 빈도가 매우 낮기 때문에 소프트웨어로 관리되도록 한다.

4.3.2 효율적인 자료 구조 관리 방안

이번 절에서는 빈번하게 접근되는 자료 구조에 대한 효율적인 관리 방안을 제시한다. 접근 빈도가 높은 자료 구조는 재사상 테이블과 재사상 진행 블록 큐가 있으며, 유지되는 정보의 양이 하드웨어로 관리될 수 있는 한도를 초과하여 증가할 수 있기 때문에 주메모리 상에서 소프트웨어로 관리된다. 이는 상위 계층과 BMS가 동시에 주메모리에 접근하는 상황을 발생시키며, 통상 DRAM으로 구현되는 주메모리에서는 메모리 접근 명령이 물리적으로 직렬화되어 수행되기 때문에 성능 저하가 초래된다. 또한 재사상 테이블의 경우, BMS에서 상위 계층 요청을 처리할 때마다 탐색 작업이 일어나는데 이는 처리 지연 시간을 심각하게 악화시킬 수 있다. 따라서 캐시와 같이 자주 참조되는 정보를 독립된 고속 메모리에 유지하여 주메모리에 대한 접근 빈도를 줄이고, 자료 구조에 대한 접근 시간을 개선하는 방안을 도입하는 것이 바람직하다.

4.3.2.1 재사상 테이블 관리 방안

재사상 테이블은 재사상된 의사 물리 블록의 사상 정보를 담고 있으며, 테이블 엔트리 (Entry)는 의사 물리 블록과 물리 블록의 쌍으로

이루어진다. FTL 요청의 블록 주소 변환 과정에서 재사상 테이블이 참조되며, 불량 블록으로 인한 재사상 작업의 결과로써 재사상 테이블의 연관되는 엔트리가 갱신된다. 성능과 밀접히 연관되는 부분은 재사상 테이블 참조이며, 의사 물리 블록 번호를 입력 받아 테이블을 탐색하는 과정을 동반한다. 테이블 탐색 결과, 의사 물리 블록과 대응되는 엔트리가 존재하면, 엔트리 내에 기록된 물리 블록 번호를 반환한다. 그러나 대응되는 엔트리가 존재하지 않는다면 해당 의사 물리 블록은 재사상되지 않았으므로, 기본 사상을 통해 물리 블록 번호를 반환한다. 대부분의 BMS 연구에서는 항등 사상 (Identity mapping)을 기본 사상으로 사용하며, 이 경우 반환된 물리 블록 번호는 의사 물리 블록 번호와 같은 값이다.

재사상 테이블은 주메모리 상에서 유지되므로 탐색 과정에서의 지연 시간이 길고, 상위 계층과 BMS 간의 주메모리 공유 문제로 인해 상위 계층의 성능이 저하될 수 있다. 따라서 탐색 과정을 가속화하고 주메모리 접근 빈도를 줄이기 위한 관리 방안이 필요하다. 이를 위해서는 하위 계층인 플래시 메모리의 불량 블록 특성과 FTL과 같은 상위 계층의 접근 특성을 고려해야 한다.

플래시 메모리에서 대부분은 정상 블록이며, 전체 블록에서 불량 블록이 차지하는 비율을 상당히 작다. 대부분의 상용 플래시 메모리 제품에서 초기 불량 블록의 수는 대개 1% 미만이며 최악의 경우에도 2%를 넘지 않는다. 또한 가격 경쟁력이 중요한 플래시 메모리 저장 장치 제품의 특성을 고려하였을 때, 물리 블록 중 대부분 사용자 데이터를 저장하는 용도로 사용되며 예비된 물리 블록은 적은 수일 것이다. 따라서 동작 중 불량 블록을 고려하더라도 전체 블록 대비 불량 블록 비율은 작을 것이다. 실례로 128 GB 용량의 플래시 메모리를 가지고 120 GB의 저장

용량을 사용하는 상용 SSD를 고려해 보면, 최대 허용 가능한 불량 블록의 비율은 6.25% 미만이다[66].

이러한 플래시 메모리의 불량 블록 특성을 고려해 보면, 대부분의 의사 물리 블록은 기본 사상되어 있을 것이며 재사상된 경우는 소수일 것이다. 특히 사용 초기에는 동작 중 불량 블록이 거의 없으므로 재사상된 의사 물리 블록의 비율은 대개 1% 미만일 것이다. 이는 의사 물리 블록을 가지고 재사상 테이블을 탐색할 때, 대부분의 경우 해당 블록을 위한 재사상 정보는 테이블 내에 존재하지 않음을 의미한다. 만약 사전에 기본 사상된 블록임을 알 수 있다면, 재사상 테이블을 탐색하는 빈도를 획기적으로 줄일 수 있다.

이와 같은 목적을 위해 재사상 블록 필터라는 자료 구조를 도입한다. 재사상 블록 필터는 접근하는 의사 물리 블록이 재사상되었는지를 알려주는 자료 구조로써, 내부적으로 재사상된 블록의 집합을 유지한다. 접근하는 블록이 재사상되지 않았다면 주소 변환 과정에서 재사상 테이블의 탐색 과정을 회피할 수 있다.

재사상 블록 필터는 블룸 필터 (Bloom filter)라는 자료 구조를 통해 효과적으로 구현될 수 있다. 블룸 필터는 비트 벡터와 같이 축약된 방식으로 집합에 대한 정보를 관리하므로 소량의 메모리만으로 구현할 수 있으며, 질의를 통해 임의의 값이 집합의 원소인지를 검사할 수 있는데 검사 작업은 상수 시간 (Constant time) 안에 완료되는 장점이 있다. 블룸 필터 내에 재사상된 블록들의 집합을 유지한다면 재사상 블록 필터의 기능이 구현된다.

특이한 점은 블룸 필터에서 긍정 오류 (False positive)가 확률적으

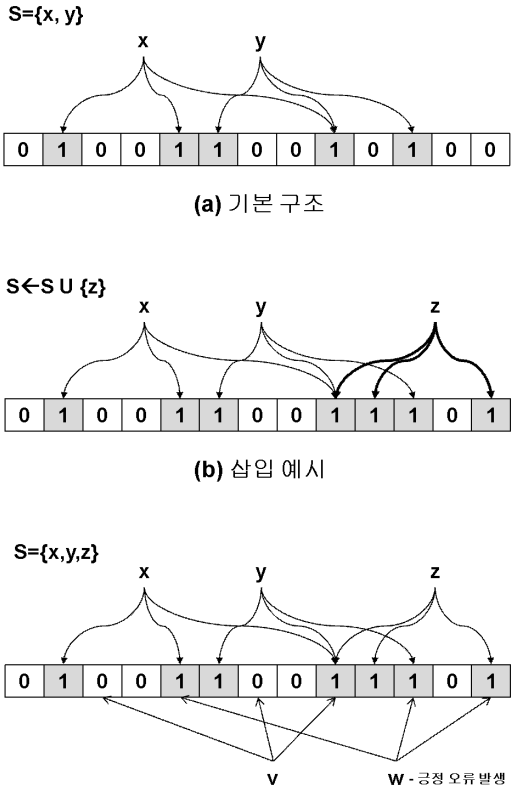


그림 27 블룸 필터 동작 예시

로 발생할 수 있다는 것인데, 이는 질의한 값이 집합에 속한다는 결과를 반환하였다고 하더라도 실제로 질의 값은 집합의 원소가 아닐 수 있음을 의미한다. 그러나 부정 오류 (False negative)는 발생하지 않으며, 이는 질의 값이 집합의 원소가 아니라는 결과를 반환하면, 해당 결과는 항상 신뢰할 수 있음을 의미한다. 또 하나의 특이한 점은 집합에 대한 원소 추가 작업은 가능하나 삭제 작업은 불가능하다는 것이다.

그림 27은 블룸 필터의 구조 및 동작 방식을 보여 준다. 블룸 필터는 내부에 비트 벡터를 가지고 있으며 각 비트는 초기값이 0이다. 새로

$m/n(=sf)$	$k=1$	$k=2$	$k=3$
1.0	63.21%	74.76%	85.79%
2.0	38.35%	39.96%	46.89%
3.0	28.35%	23.68%	25.26%
4.0	22.12%	15.48%	14.69%
5.0	18.13%	10.87%	9.19%
6.0	15.35%	8.04%	6.09%
7.0	13.31%	6.18%	4.24%
8.0	11.75%	4.89%	3.06%
9.0	10.52%	3.97%	2.28%
10.0	9.52%	3.29%	1.74%
11.0	8.69%	2.76%	1.36%
12.0	7.99%	2.36%	1.08%
13.0	7.40%	2.03%	0.86%
14.0	6.89%	1.77%	0.72%
15.0	6.45%	1.56%	0.60%

그림 28 블룸 필터 긍정 오류 확률

운 값을 원소로 추가하는 경우, 복수 개의 해시 함수를 통해 해당 원소에 대응되는 비트 위치를 구하고, 각 위치의 비트 값을 1로 설정한다. 그림 27 (a)는 두 개의 원소 x, y로 구성되어 있는 블룸 필터의 내부 구조를 보여 준다. 세 가지 서로 다른 해시 함수를 사용하고 있기 때문에 x, y 각각은 세 개의 비트에 대응된다. 주목할 점은 x, y가 동일한 비트에 대응될 수 있다는 점이다. 새로운 원소 z가 추가되면 그림 27 (b)와 같이 비트 벡터 내의 1의 개수가 늘어난다. 그림 27 (c)는 질의 과정을 나타낸다. 질의 값이 x인 경우 해시 함수에 대응되는 비트 값이 모두 1이므로 집합에 포함된다는 결과를 반환한다. 질의 값 v에 대해서는 대응되는 비트 값 중 0인 것이 있기 때문에 집합에 포함되지 않는다는 결과를 반환한다. 질의 값 w는 실제로 집합에 포함되지 않지만 대응 비트들이 모두 1이므로 긍정 오류에 해당한다. 긍정 오류는 집합 내의 서로 다른 원소가 대응 비트들을 공유할 수 있기 때문에 발생하며, 집합 내의

원소 수에 비해 비트 벡터의 크기가 클수록 긍정 오류 확률은 낮아진다. [67]에서 예측한 긍정 오류 확률은 그림 28과 같다. m은 비트 벡터의 길이, n은 집합 내의 원소의 개수, sf는 크기 인자 (Scale Factor=m/n), k는 사용하는 해시 함수의 개수를 나타낸다.

블룸 필터의 특성은 재사상 블록 필터를 구현하기에 적합하다. 블룸 필터를 통해 재사상된 블록의 집합을 소량의 메모리로 유지할 수 있으며, 상수 시간 안에 집합에 대한 포함 여부를 검사할 수 있다. 또한 BMS의 구현 특성 상, 기본 사상된 의사 물리 블록이 재사상되는 경우는 가능하나 재사상된 블록이 기본 사상된 블록으로 돌아가는 경우는 발생하지 않는다. 따라서 재사상된 블록 필터에서는 삽입 연산만 발생할 뿐 삭제 연산은 발생하지 않으며 이는 블룸 필터의 동작 특성과 잘 부합된다. 마지막으로 기본 사상된 블록에 대해 재사상되었다고 알려 주는 긍정 오류가 발생할 수 있으나, 이는 불필요한 재사상 테이블 탐색 작업을 야기할 뿐 정확성에는 영향을 미치지 않는다. 긍정 오류 확률은 소량의 메모리를 통해 충분히 낮은 수준을 유지할 수 있다.

그림 29는 재사상 블록 필터를 활용한 주소 변환 과정을 보여 준다. FTL 요청의 의사 물리 블록 번호를 입력받으면 재사상 블록 필터를 통해 재사상된 블록인지 검사한다. No라는 결과를 받으면 기본 사상된 블록이므로 재사상 테이블을 탐색하지 않고 주소 변환 과정을 완료하게 된다. Yes라는 결과를 받으면, 재사상된 블록이거나 긍정 오류로 인해 재사상된 블록 여부를 정확하게 알 수 없는 상황이므로 재사상 테이블을 탐색하여 주소 변환 과정을 수행하게 된다.

블룸 필터를 통한 구현 방식의 효용성은 재사상된 의사 물리 블록

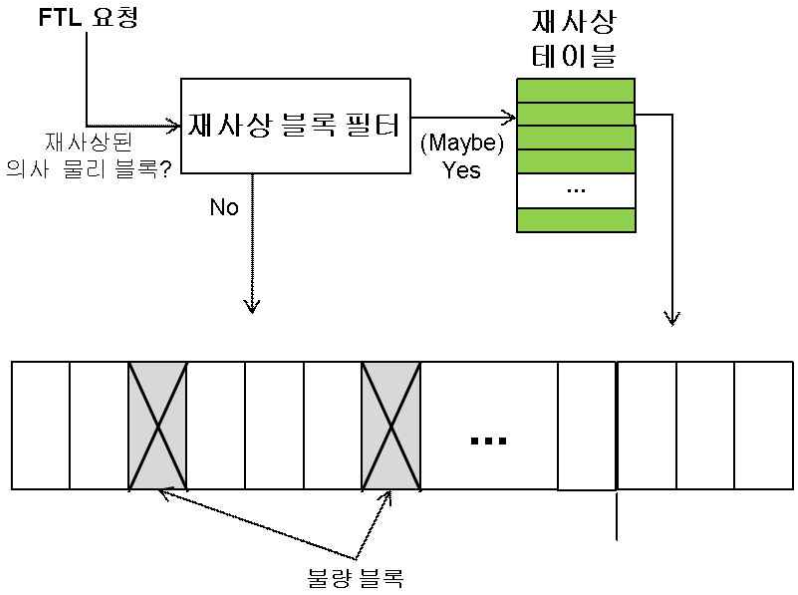


그림 29 재사상 블록 필터의 활용

의 개수와 블룸 필터를 구현하는 비트 벡터의 크기에 의해 결정된다. 재사상된 의사 물리 블록의 경우 블룸 필터의 반환 결과가 Yes이므로 항상 재사상 테이블을 탐색하게 된다. 그러나 최악의 경우에도 재사상된 블록의 비율은 적을 것이므로 재사상 테이블 탐색 비용은 효과적으로 감소한다. 기본 사상된 블록들은 확률적으로 재사상 테이블을 탐색하게 되는데, 긍정 오류 확률에 의해 탐색 빈도가 결정된다. 긍정 오류 확률은 블룸 필터 내부의 비트 벡터를 충분히 확보함으로써 목표 수준까지 낮출 수 있다. 예를 들어 전체 의사 물리 블록의 수가 100,000개이고 재사상된 블록이 전체의 2%인 2,000개라고 가정하자. 블룸 필터의 비트 벡터 길이를 30,000 약 (3.7 KB)으로 설정하고 3개의 해시 함수를 사용한다면, 그림 28에서 sf값이 15이므로 긍정 오류 확률은 0.6% 수준이 된다. 만약 재사상된 블록의 수가 증가하더라도 전체 블록 대비 재사상된 블록

의 비율은 최대 6% 수준이므로 [66], 최악의 경우에도 긍정 오류 확률은 일정 수준 이하에 머무르게 된다. (현재 예에서 재사상 블록 비율이 6%가 되면 그림 28에서 sf값이 5가 되어 긍정 오류 확률은 9.19%가 된다)

블룸 필터의 비트 벡터는 고속으로 동작하는 SRAM을 통해 구현하는 것이 바람직한데, 저장 장치의 용량이 증가함에 따라 필요한 SRAM의 크기는 선형적으로 증가할 것이다. 그러나 반도체 제조 공정이 미세화됨에 따라 ASIC이나 FPGA에서 사용 가능한 SRAM의 양도 함께 증가하고 있기 때문에 저장 장치의 용량이 증가하더라도 블룸 필터를 위한 SRAM을 충분히 확보할 수 있을 것이다. 앞선 예에서 의사 물리 블록 공간의 용량이 1 TB (테라바이트)로 증가했다고 하자. 단일 블록의 크기를 1 MB라고 가정하면 전체 의사 물리 블록의 수는 약 10배로 증가하게 되며 이에 따라 블룸 필터에서 사용하는 SRAM의 크기도 37 KB로 증가하게 된다. 이 정도의 SRAM은 충분히 사용 가능한 크기인데, 현재의 ASIC이나 FPGA에서는 수 MB ~ 수십 MB의 SRAM을 사용할 수 있기 때문이다. (Xilinx 사의 FPGA인 Virtex5 XC5VFX130T[69]의 경우 내부 SRAM 용량을 합하면 9 MB 이상이며, 최근 출시된 FPGA들은 이보다 훨씬 큰 SRAM 용량을 사용할 수 있다) 따라서 저장 장치 용량이 테라바이트 급으로 증가하더라도 블룸 필터를 적용하는 것은 가능할 것이다.

블룸 필터는 재사상 테이블 탐색 과정을 효과적으로 회피할 수 있지만, 재사상된 블록이나 긍정 오류가 발생하는 블록들에 대한 주소 변환의 경우, 재사상 테이블 탐색이 확정적으로 발생하는 문제가 있다. 로그 형식으로 의사 물리 블록들을 관리하는 FTL의 동작 특성을 미루어

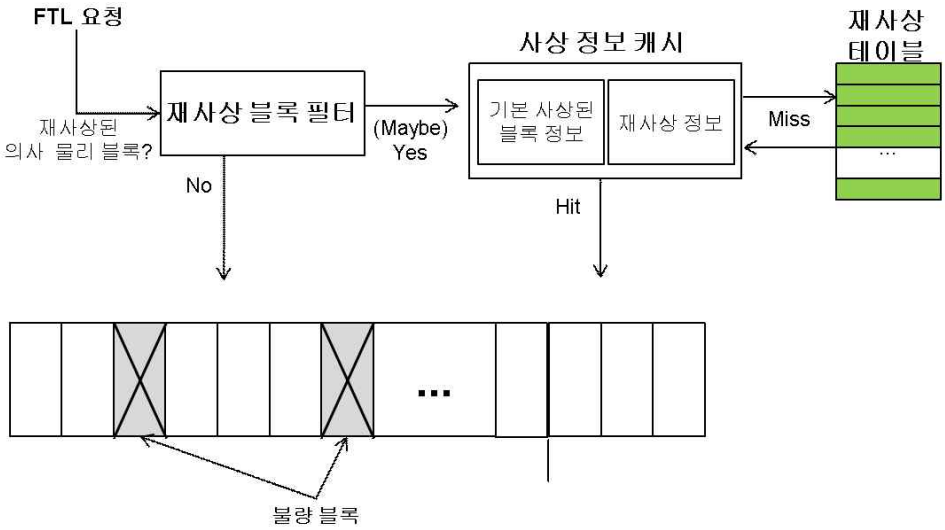


그림 30 사상 정보 캐시의 활용

생각해 보면, 짧은 시간 구간 안에서 한정된 수의 의사 물리 블록에 집중적으로 요청이 발생할 것을 예상할 수 있다. 만약 재사상된 블록에 대한 접근 요청이 연속적으로 발생하면, 짧은 시간 안에 재사상 테이블 탐색 작업을 과도하게 수행하게 되며, 일시적으로 전체 저장 장치의 성능이 하락하는 현상이 발생할 것이다. 이러한 문제점은 사상 정보에 대한 캐시를 추가로 도입하여 해결할 수 있다. 그림 30은 재사상 블록 필터에 사상 정보 캐시를 추가해서 사용하는 경우의 동작 방식을 보여 준다. 재사상 블록 필터에서 Yes를 반환하는 경우, 사상 정보 캐시에 접근하여 사상 정보를 찾게 된다. 사상 정보 캐시에는 재사상 테이블 탐색을 통해 최근 참조된 사상 정보가 유지된다. 구현 비용을 줄이기 위해 캐시를 재사상 정보 영역과 기본 사상 블록 영역으로 구분하는데, 전자는 재사상된 의사 물리 블록 번호와 이에 대응하는 물리 블록 번호를 함께 유지하지만 후자는 의사 물리 블록 번호만 유지하기 때문에 캐시 용량을 줄일

수 있다. 캐시 히트 (Cache hit)인 경우에는 재사상 테이블 탐색 없이 주소 변환 과정을 완료하게 된다. 캐시 미스 (Cache miss)인 경우에는 재사상 테이블을 탐색하여 사상 정보를 캐시에 적재하고, 캐시 히트 시의 처리 과정을 수행하여 주소 변환 과정을 완료한다. 이와 같은 사상 정보 캐시는 소량의 메모리만으로 재사상 탐색 비용을 효과적으로 줄일 수 있는데, 이는 FTL 요청이 의사 물리 블록 수준에 대한 시간 지역성 (Temporal locality)과 공간 지역성 (Spatial locality)을 가지기 때문이다.

재사상 블록 필터와 사상 정보 캐시의 내용은 재사상 작업이 수행되는 경우 내용 갱신이 일어나는데, 이 때 주소 변환 참조가 동시에 발생하더라도 항상 최신의 주소 변환 결과를 반환해야 한다. 그림 26의 BMS 구조에 의하면 재사상 작업 중에는 해당 의사 물리 블록에 대한 주소 변환이 일어나지 않는다. 이는 갱신이 완료될 때까지 동일 사상 정보에 대한 참조는 발생하지 않음을 의미하며, 이러한 속성에 의해 주소 변환 결과의 최신성은 보장된다.

4.3.2.2 재사상 진행 블록 큐 관리 방안

재사상 진행 블록 큐는 재사상이 진행 중인 의사 물리 블록들의 리스트이다. FTL 요청 처리 중 불량 블록이 발생하면 요청 대상인 의사 물리 블록은 재사상 진행 블록 큐에 추가되며 재사상 작업 완료 후 큐에서 제거된다. 새로운 FTL 요청이나 플래시 제어기 응답이 발생할 때마다 진행 중인 재사상 작업과의 의존성을 검사하기 위해 재사상 진행 블록 큐가 탐색된다. 그러나 재사상 작업은 드물게 발생하기 때문에 대부분의 경우 재사상 진행 블록 큐는 비어 있거나 적은 수의 요청들만 존재

할 것이다. 따라서 재사상 진행 블록 큐는 소량의 고속 메모리를 통해 구현할 수 있으며, 재사상 진행 중인 블록의 수가 고속 메모리 용량을 초과하는 극단적인 경우에만 주메모리 공간을 할당받아 요청들을 임시로 저장할 수 있다.

제 5 장 검증 및 성능 평가

5.1 시뮬레이션 기반 정확성 검증

X-BMS[9]의 자료 구조 및 재사상 알고리즘을 기반으로 병렬화된 BMS를 구현하여 그림 31의 시뮬레이터 기반 환경에서 동작의 정확성을 검증하였다. 시뮬레이션 환경은 BMS를 동작시키기 위한 구동부와 BMS의 실행 결과와 내부 상태를 분석하여 동작을 검증하는 검증부로 나누어진다. BMS의 하단에 위치한 플래시 메모리 시뮬레이터[68]는 X-BMS에서 가정한 페이지 동작 모델에 따라 플래시 메모리의 동작을 모사한다. 또한 설정에 따라 칩의 개수 및 페이지 크기나 블록 개수와 같은 칩의 특성 값들을 변경할 수 있으며, 무순서 플래시 메모리 제어기의 스케줄링 방식을 모사하여 다수의 칩이 동시에 구동되도록 한다. 작업 부하 생성기는 FTL 요청을 사용자의 설정에 따라 생성하여 BMS로 전달한다. 불량 블록 생성기는 플래시 메모리 연산중에 지우기 및 쓰기 오류를

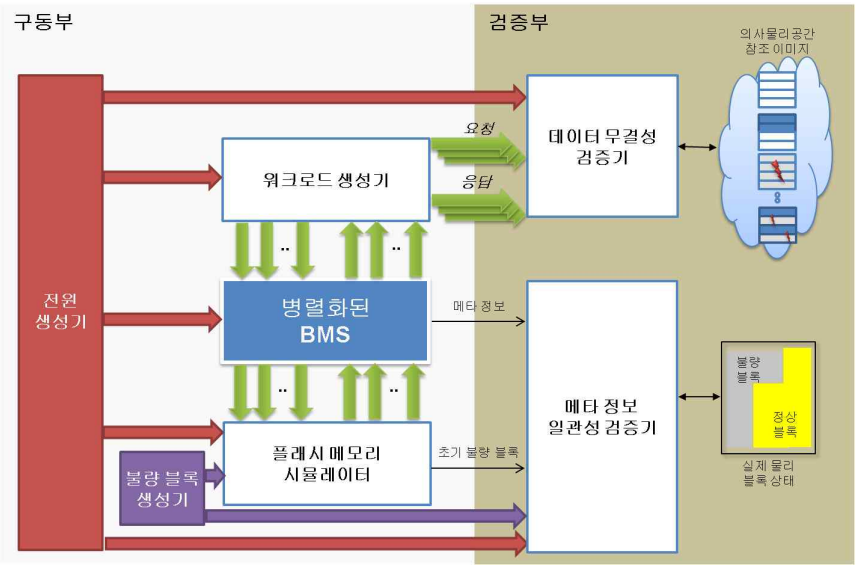


그림 31 시뮬레이션 기반 검증 환경

발생시키며, 전원 오류 생성기는 플래시 메모리 연산과 비동기적으로 전원 오류를 발생시킨다. 오류 발생 확률은 사용자가 설정할 수 있다.

검증부는 BMS가 상위 계층에 제공하는 인터페이스를 통해 데이터 일관성 (Data coherence)을 검증하는 데이터 일관성 검증기와 BMS의 내부 상태를 추적하여 일관성을 검증하는 메타 정보 일관성 검증기로 나누어진다. 데이터 일관성 검증기는 작업 부하 생성기가 BMS와 주고 받은 요청과 응답 결과를 분석하여 데이터 일관성을 검사한다. 데이터 일관성이란 읽기 요청에 대한 응답으로 최신의 데이터를 반환해야 한다는 BMS 인터페이스의 정확성 기준이다. 성공적으로 지워지거나 쓰여진 의사 물리 페이지의 경우, 최신의 데이터는 명확하게 정의된다. 그러나 지우기나 쓰기 연산 중 전원 오류로 인해 연산이 중단된 페이지의 상태는 가정하고 있는 페이지 동작 모델에 의존적이다. X-BMS의 정확성 증명에 사용된 페이지 동작 모델에 따르면 쓰기 연산이 미완료된 페이지의 상태는 쓰기 연산이 전혀 진행되지 않은 상태, 쓰기 연산이 완료된 것과 동일한 상태, 임의의 중간 상태를 가질 수 있으며, 각각의 상태에 대하여 읽기 연산의 결과로 모든 비트가 1인 값, 쓰기 의도된 데이터 값, 쓰기 의도되지 않은 데이터로서 ECC (Error Correction Code)에 의해 비트 오류가 검출되는 값을 반환할 수 있다. 지우기 연산이 미완료된 경우에 대해서도 비슷한 방법을 통해 가능한 읽기 연산 결과를 도출할 수 있다. 데이터 일관성 검증기는 이러한 데이터 일관성 규칙을 만족시키는 의사 물리 블록 공간의 이미지를 내부적으로 유지하며, BMS에서 반환한 데이터를 해당 이미지 상의 데이터와 비교하여 데이터 일관성을 검증한다.

메타 정보 일관성 검증기는 두 가지 기준에서 BMS의 메타 정보 일

관성을 검사한다. 첫 번째 기준은 불량 블록에 대한 무결성인데, 이는 BMS가 정상 블록을 불용 블록으로 등록하지 않으며, 한 번 불용 블록으로 등록되면 다시는 사용하지 않는다는 것이다. 이를 위해 내부 오류 생성기가 발생한 오류에 따라 실제 불량 블록과 정상 블록을 파악하고, BMS 메타 정보로부터 불용 블록에 대한 정보를 추출하여 실제 불량 블록 정보와 비교하여 무결성을 검사한다. 두 번째 기준은 상호 배제와 전체 포괄에 대한 무결성이며 이는 모든 물리 블록들이 데이터 블록, 예비 블록, 불용 블록, 시스템 블록 중 하나의 집합에 포함되고 블록 집합 간에는 교집합이 없음을 나타낸다. BMS 메타 분석 분석을 통해 이러한 기준이 만족되는지 검사한다.

시뮬레이션 환경을 구성하는 각각의 모듈은 리눅스 운영체제에서 제공하는 스레드 (Thread) 기능을 통해 병렬적으로 구동하였다. 실제 상황보다 높은 빈도로 오류가 발생하도록 오류 확률을 설정하였으며, 임의로 동기적/비동기적 오류가 중첩하여 발생하는 상황을 만들기 위해 오류 확률을 동적으로 변경하였다. 즉, 한 번 오류가 발생하면 일정한 시간 구간 안에서는 오류 확률을 초기 설정보다 높은 값으로 변경하여, 중첩된 오류의 가능성을 높였다. 시뮬레이션 결과, 병렬화된 BMS가 정확히 동작함을 확인하였다.

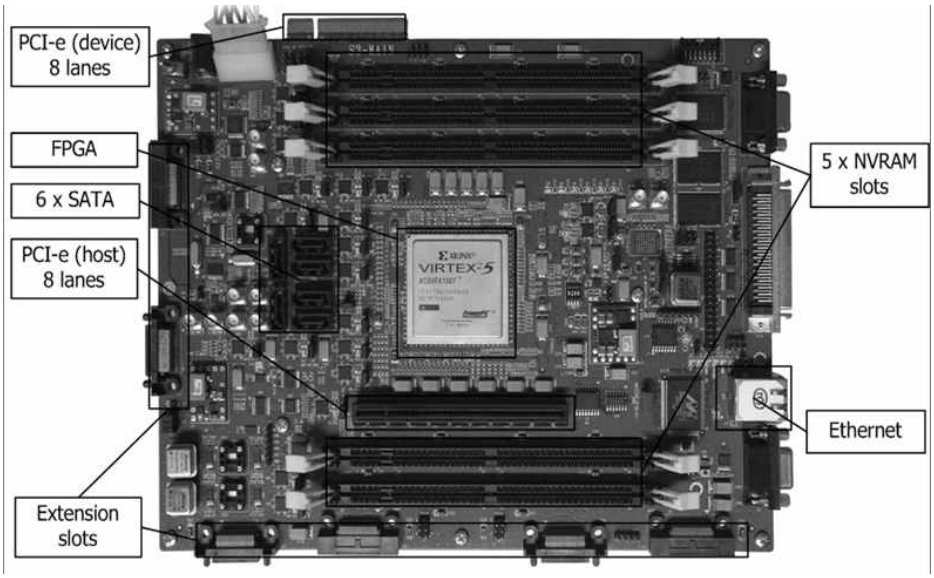


그림 32 평가 플랫폼 보드 [6]

그림 33 다양한 NAND 모듈 [6]

5.2 성능 평가

5.2.1 실험 환경

병렬화된 BMS를 구현하여 무순서 플래시 메모리 제어기[6]가 구축된 개발 보드 (그림 32) 상에서 성능을 평가하였다. 개발 보드는 Xilinx 사의 FPGA (Virtex5 XC5VFX130T)[69]를 탑재하고 있는데, FPGA 내부에는 무순서 플래시 메모리 제어기 뿐만 아니라 PowerPC440 프로

Parameters	Value
Page read	56 µs - 62 µs
Page program	524 µs - 1062 µs
Block erase	2 ms
I/O cycle time	30 ns (min)
Page size	4KB + spare area
Block size	128 pages

그림 34 플래시 메모리 매개 변수[6]

세서, AXI 시스템 버스, SRAM의 일종인 block RAM, 각종 메모리 제어기, UART (Universal Asynchronous Receiver/Transmitter) 와 같은 실험에 필요한 기본적인 요소들을 구축하였다. 개발 보드에는 5 개의 확장 슬롯이 있으며 각 슬롯은 2개의 플래시 메모리 버스를 지원한다. 4개의 슬롯에 그림 33과 같은 NAND 플래시 메모리 모듈을 장착하였는데, 각 모듈은 2개의 플래시 메모리 버스로 구성되며 개별 버스에는 8개의 칩들이 연결되어 있다. 플래시 메모리 칩은 주요 특성 값들은 그림 34와 같다. 그 밖에 개발 보드에는 주메모리로 사용할 수 있는 128MB의 mobile DDR 메모리를 장착하여 BMS 소프트웨어 및 테스트 코드를 구동하는 용도로 활용하였다.

병렬화된 BMS는 전체를 소프트웨어로 구현한 버전 (이하 SW BMS)과 자주 사용되는 기능을 하드웨어 BMS로 구현한 버전 (이하 HW BMS)을 나누어서 평가하였다. 또한 하드웨어 모듈들은 verilog HDL을 이용하여 구현하였으며, Xilinx 사의 ISE 툴을 이용하여 합성한 결과 FPGA 자원 사용량은 표 35와 같다. HW BMS의 자원 사용 효율을 파악하기 위해 단일 플래시 메모리 버스에 대한 무순서 플래시 제어기의

	무순서 플래시 제어기 (단일 버스)	HW BMS
Slice-registers	6,253 (7%)	1,098 (1%)
Slice-LUTs	11,453 (13%)	2,687 (3%)
Occupied slices	4,292 (20%)	1,078 (5%)
Block RAM	59 (19%)	14 (4%)

그림 35 FPGA 자원 사용량

자원 사용량을 [6]에서 발췌하여 그림 35에 함께 요약하였다. 무순서 플래시 메모리와 HW BMS는 동일한 FPGA를 대상으로 합성되었기 때문에 직접적인 자원 비교가 가능하다. 비교 결과 HW BMS는 플래시 제어기에 비해 FPGA slice 측면에서 약 75 % 적은 자원을 사용하고 있다. 플래시 버스의 개수에 따라 무순서 플래시 제어기의 자원 사용량도 증가하지만, HW BMS의 자원 사용량은 버스 개수와 무관하다. 따라서 다수의 플래시 버스를 사용하는 저장 장치의 경우 HW BMS의 자원 소모량은 플래시 제어기에 비해 훨씬 작을 것이다. 이와 같은 사실은 HW BMS가 상이한 자원 요구 사항을 가지는 다양한 저장 장치에 적용될 수 있음을 보여 준다.

그림 36은 BMS를 평가하는 실험 환경의 구성을 보여준다. 작업 부하 생성기는 FTL과 같은 상위 계층 요청을 생성하며, 평가 대상을 선택하여 요청을 전달한다. 성능 평가 대상에는 HW BMS 및 SW BMS 뿐만 아니라 플래시 제어기도 포함되는데, 이는 BMS 계층화로 인한 성능 저하를 분석하기 위한 것이다. 플래시 제어기에는 주메모리와 플래시 메모리 사이에 데이터를 전송할 수 있도록 DMA 기능을 추가하였다. 오류 생성기는 플래시 제어기로 향하는 요청의 인자 (Argument)들을 변경하

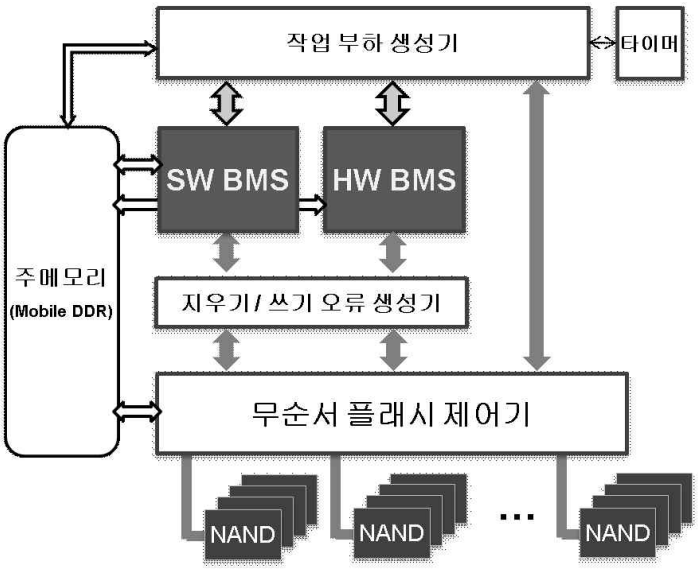


그림 36 실험 환경 구성

여 지우기 및 쓰기 오류가 발생한 효과를 일으킨다. 지우기 오류의 경우, 요청 대상 블록이 아닌 미리 할당해 놓은 블록에 수행하고 오류를 반환하도록 하였다. 대상 블록은 실제로 지우기 연산이 수행되지 않았기 때문에, 상위 계층의 관점에서는 지우기 오류가 발생한 것처럼 보일 것이다. 쓰기 오류의 경우, 의도하지 않은 데이터가 쓰여지도록 쓰기 요청을 변경함으로써 오류의 효과를 모사하였다. 타이머 (Timer)는 실행 시간을 측정하여 평가 대상의 요청 처리율 (Throughput)과 응답 시간 (Response time)을 구하기 위한 용도로 사용된다.

5.2.2 성능 평가

무순서 플래시 메모리 제어기 상에서 SW BMS와 HW BMS를 구동하여 성능을 평가하였다. 다양한 용량의 저장 장치에 대한 적용 가능성을 분석하기 위하여, 플래시 메모리 버스의 개수 및 버스 당 칩의 개수

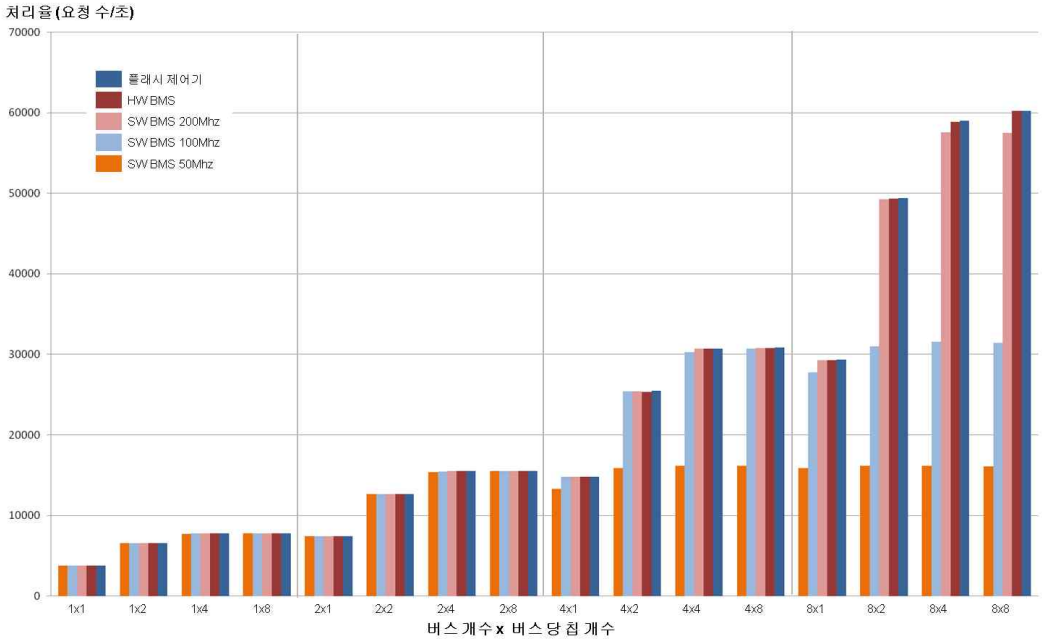


그림 37 버스 및 칩 개수에 따른 요청 처리율 (정상 동작 상황)

를 변경하여 실험을 수행하였다. 작업 부하는 지우기, 쓰기, 읽기 요청이 조합된 합성 작업 부하를 사용하였다.

5.2.2.1 정상 동작 상황에서의 성능 평가

저장 장치에서 불량 블록이 발생하는 경우는 드물기 때문에, 불량 블록이 발생하지 않는 상황 하에서 성능을 평가하였다. 그림 37은 SW BMS, HW BMS 및 플래시 메모리 제어기의 성능을 요청 처리율 (단위 시간 당 처리한 요청 수) 측면에서 보여 준다. SW BMS의 성능은 프로세서에 대한 점유율에 의해 영향을 많이 받을 것이므로 프로세서 클록을 변화시켜 점유율이 변동한 효과를 발생시켰다. HW BMS는 재사상 블록 필터, 사상 정보 캐시를 각각 2KB씩 할당하였으며, 이 때 재사상 테이

블의 탐색 회피율은 99% 이상이었다. 또한 HW BMS에 인가된 클록 주파수는 100 Mhz인데, 이는 플래시 제어기와 BMS 간의 인터페이스 클록과 동일한 값이다. 작업 부하에서 지우기, 쓰기, 읽기 요청의 비율을 1:128:128로 설정하였는데, 이는 플래시 메모리 블록 내의 페이지 수가 128개이기 때문이다.

그림 37을 보면 플래시 버스의 개수가 1 또는 2인 경우에는 구현 방식과 상관 없이 BMS의 성능이 플래시 제어기의 성능과 비슷한 수준을 유지하였다. 플래시 제어기에 대한 성능 저하율은 모든 경우에 대해서 0.01% 미만이었으며 측정 오차 범위 이내의 값이다. 이는 저용량의 저장 장치에서는 SW로 구현된 BMS라고 하더라도 성능 저하가 거의 없음을 의미한다. 버스가 4개로 증가하는 경우 50Mhz 클록을 사용하는 경우의 SW BMS 성능이 플래시 제어기에 비해 급격히 떨어지는 것을 볼 수 있는데, 성능 포화 현상이 발생했기 때문이다. 플래시 제어기와 비교하여 성능 저하율은 버스 당 칩 개수에 따라 37%에서 47% 수준이었다. 그러나 100 Mhz이상의 클록 속도에서는 SW 구현이라도 성능 저하가 거의 없었다.

8개 버스에서는, 100Mhz 클록을 사용하는 SW BMS의 성능 저하가 눈에 띄게 증가하였으며, 성능 저하율은 37%에서 47% 수준이었다. 이는 4개 버스에서 50Mhz의 SW BMS와 비슷한 성능 저하율인데, 플래시 버스 개수가 2배로 증가하면서 플래시 제어기의 성능이 2배로 증가했으며, 100Mhz SW BMS가 50Mhz인 경우에 2배의 성능을 낼 수 있기 때문에 성능 저하율 비슷한 것으로 판단된다.

200Mhz SW BMS의 성능은 버스 개수가 8개 이고 버스 당 칩의

개수가 4개 이상인 경우에 포화되었으며, 성능 저하율은 5% 미만이었다. SATA 인터페이스를 사용하는 저가의 SSD에도 통상 8개 이상의 플래시 버스를 사용하기 때문에, SW BMS를 사용한다면 성능 저하가 발생할 것이며, PCI express 인터페이스를 사용하는 서버용 SSD는 더욱 많은 버스를 사용하므로 SW BMS를 적용할 수 없을 것이다. 더욱이 FTL과 같은 상위 계층이 프로세서 점유율의 상당 부분을 차지할 것이므로, SW BMS를 SSD와 같은 대용량 저장 장치에 적용하기는 현실적으로 불가능할 것이다.

HW BMS의 성능 저하율은 8개의 버스에서도 0.01% 미만이었으며, 이는 오차 범위 이내이므로 실질적으로 성능 저하가 없음을 의미한다. HW BMS의 성능을 심도 있게 분석하기 위해 버스의 개수를 더 늘릴 필요가 있으나 실험 환경의 제약으로 인해 분석을 수행하지 못하였다. 클록 사이클 수를 통해 계산한 HW BMS의 요청 처리 지연 시간은 약 0.2 us (약 20 사이클) 이며, 이는 플래시 제어기가 입력된 요청을 플래시 메모리 칩에 전달하는 시간보다 짧은 시간이다. 더욱이 HW BMS에서 사용하는 클록은 일반 ASIC 회로에서 사용하는 클록보다 훨씬 낮은 속도이다. 따라서 버스 개수가 증가하더라도 HW BMS로 인한 성능 저하는 경미할 것으로 예측할 수 있다.

그림 38은 8개 버스 및 버스 당 8개 칩을 사용하는 경우의 응답 시간의 분포를 보여준다. HW BMS와 200 Mhz의 SW BMS는 성능 저하율이 거의 없으므로 플래시 제어기와 비슷한 개형의 응답 시간 분포를 가진다. 그러나 성능 저하율이 큰 50, 100 Mhz의 SW BMS는 플래시 제어기로 전송되지 못하고 밀린 요청 (Backlog)에 의한 효과가 증가하여 응답시간이 넓게 분포하는 양상을 가진다. 평균 응답 시간 증가율은

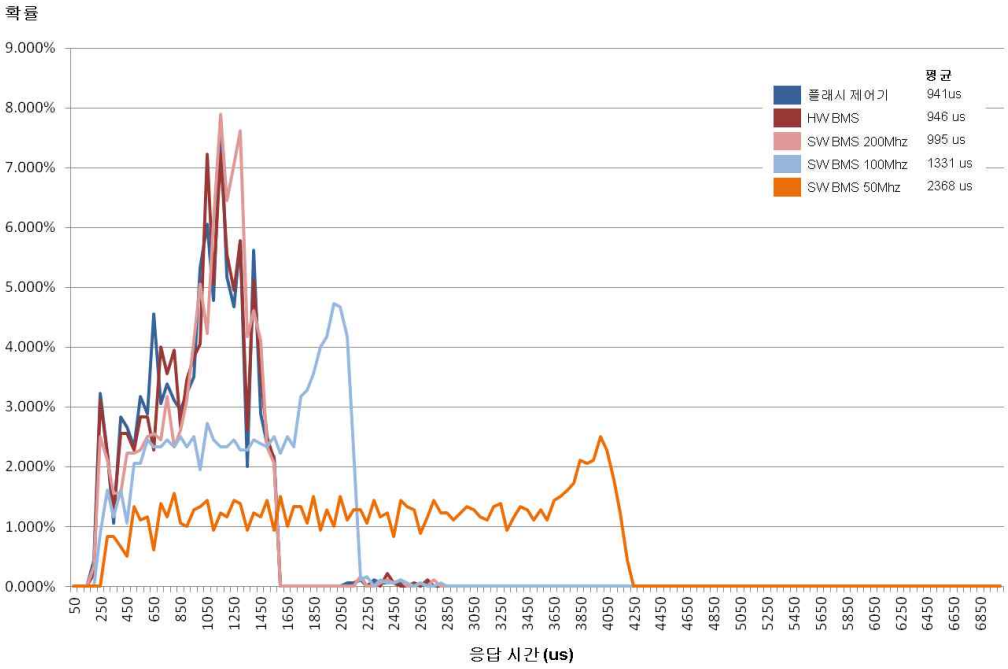


그림 38 응답 시간 분포 (정상 동작 상황)

50, 100 Mhz의 SW BMS의 경우 각각 151%, 42% 수준이었으며, 200 Mhz SW는 6% 수준이었다. HW BMS는 오차 범위 이내로써 실질적인 응답 시간 증가가 없었다.

5.2.2.2 불량 블록이 발생하는 상황에서의 성능 평가

불량 블록이 발생하는 경우를 고려하기 위해 인위적으로 지우기 및 쓰기 오류를 발생시켜서 저장 장치의 성능을 평가하였다. 평가 대상 플래시 메모리는 내구성 수치가 10,000 이며, 이는 블록에 대하여 10,000 번의 지우기 및 쓰기 반복을 보장함을 의미한다. 그러나 이러한 내구성 수치를 고려하여 불량 블록을 발생시키는 것은 적절하지 않은데, 상위 계층의 마모도 평준화를 사용하기 때문에 저장 장치가 일정 이상 노후화

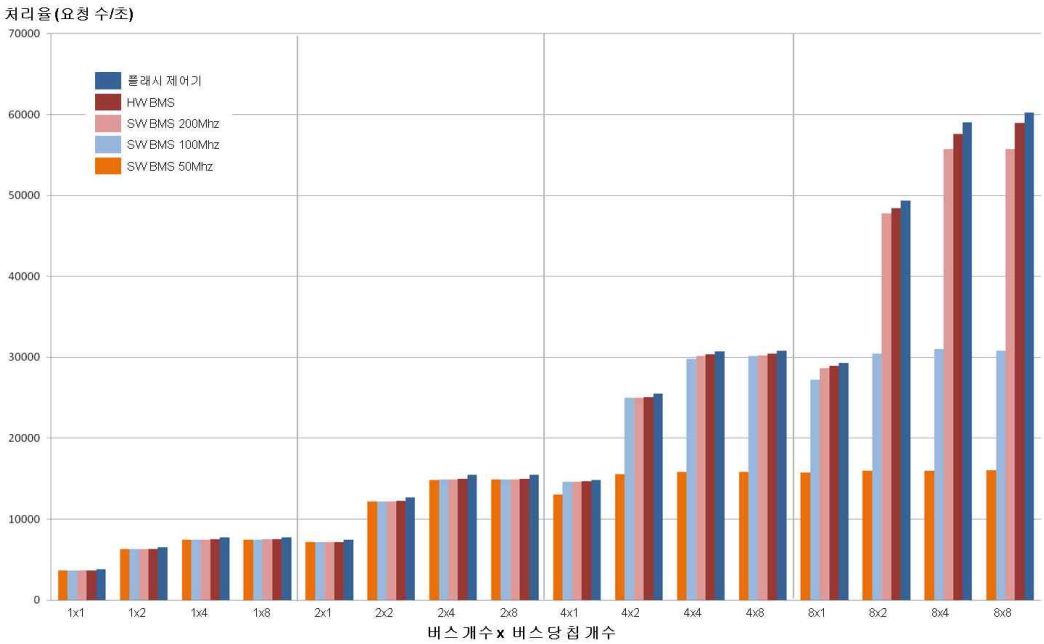


그림 39 버스 및 칩 개수에 따른 요청 처리율

(불량 블록 발생 빈도 - 지우기 및 쓰기 반복 횟수 500회 당 1번)

가 진행된 시점에서 불량 블록이 집중적으로 발생할 것이기 때문이다. 따라서 내구성 수치에 비해 자주 불량 블록이 발생하도록 설정하여 실험을 수행하였다.

그림 39는 500 번의 지우기 및 쓰기 반복을 할 때마다 불량 블록이 발생하는 경우의 성능을 보여 준다. 불량 블록 처리로 인해 추가의 플래시 메모리 연산이 발생하였기 때문에 전체적으로 BMS의 성능이 저하되었다. 그러나 추가된 연산의 비율이 전체 연산에 비해 상당히 작기 때문에 성능 저하 폭은 약 3% 미만이었다.

그림 40은 100 번의 지우기 및 쓰기 반복마다 불량 블록이 발생하

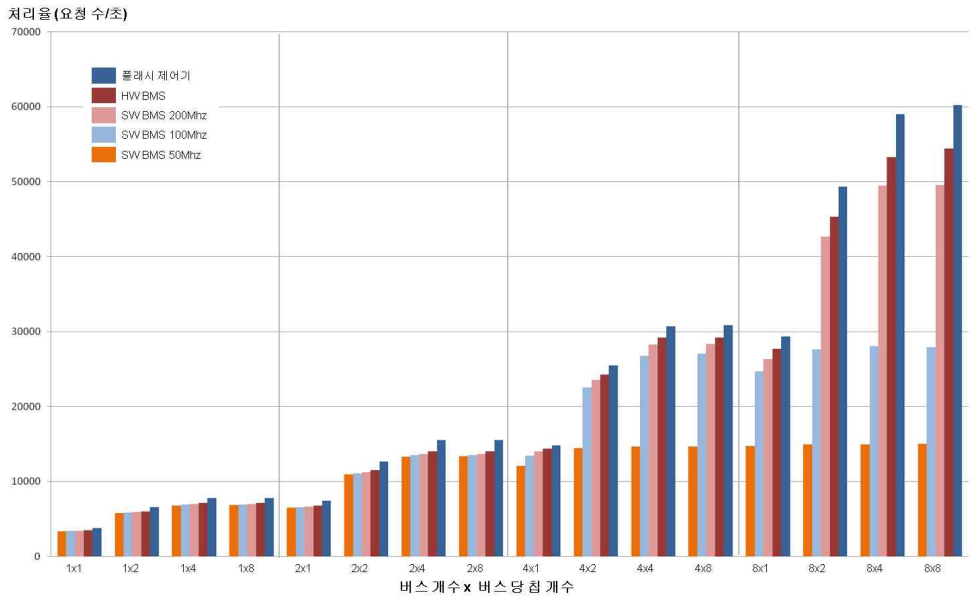


그림 40 버스 및 칩 개수에 따른 요청 처리율

(불량 블록 발생 빈도 - 지우기 및 쓰기 반복 횟수 100회 당 1번)

는 경우의 성능이다. 불량 블록 빈도가 높아졌기 때문에 그림 39에 비해 성능 저하율이 증가하였다. 그러나 추가된 연산의 비율이 여전히 전체 연산에 비해 상당히 작기 때문에 성능 저하 폭은 약 10% 미만으로 비교적 작은 수치였다.

위의 두 실험을 통해 불량 블록이 발생하더라도 저장 장치 성능 저하폭은 크기 않음을 알 수 있다. 만약 불량 블록 발생 확률이 커진다면 성능 저하폭도 더욱 증가할 것이나, 그와 같은 경우는 저장 장치의 노후화가 진행된 경우로써 장치가 교체될 가능성이 높은 경우이다. 따라서 대부분의 경우에 불량 블록이 발생 확률은 크지 않을 것이며, 불량 블록 처리로 인한 성능 저하 폭도 작을 것이다.

제 6 장　결론 및 향후 연구 과제

본 논문에서는 다수의 플래시 메모리 연산 요청들이 병렬적으로 수행되는 시스템에서 정확한 동작을 보장하면서도 요청 처리 지연을 최소화할 수 있는 불량 블록 관리 기법 (BMS)을 제안하였다. 이를 위하여 불량 블록 관리와 연관된 의존성 문제를 체계적으로 정리하고 해결 방법을 제시하였다. BMS에서 요청 간의 의존성은 연산 중 불량 블록이 발생하는 예외 상황과 이를 처리하는 구체적인 알고리즘에 종속적이다. 본 논문에서는 BMS의 인터페이스 및 동작에 대한 최소한의 가정을 도입하여 의존성 문제를 규명하였고, FTL과 같은 상위 계층의 동작 특성을 고려하여 대부분의 경우에 요청 처리 지연이 발생하지 않는 의존성 해결 규칙을 마련하였다. 이러한 의존성 해결 규칙은 특정 BMS 구현에 한정되지 않고, 인터페이스 및 동작 가정을 만족하는 모든 BMS 구현에 적용될 수 있는 장점이 있다.

정리한 의존성 문제는 다음과 같다.

- 데이터 의존성: 동일 의사 물리 블록 또는 페이지의 내용을 변경하거나 읽는 순서에 의해 발생하는 의존성

- 쓰기 순서 제약에 의한 의존성: 플래시 메모리의 오름차순 쓰기 제약에 의해 발생하는 의존성

- 재사상에 의한 의존성: 불량 블록 처리로 인해 BMS의 메타 데이터를 갱신함에 따라 발생하는 의존성

의존성 문제를 해결함에 있어 구현 복잡도를 줄이기 위해 데이터 의

존성은 계층적입 접근 방법을 통해 해결하였다. 그리고 다음과 같은 실행 규칙을 도입하여 쓰기 순서 및 재사상에 의한 의존성을 해결하였다.

실행 규칙1: 동일 의사 물리 블록으로 향하는 두 FTL 요청에 대하여, 플래시 메모리 제어기로 연산 실행 요청을 순차적으로 전달한다.

실행 규칙2: 동일 의사 물리 블록으로 향하는 두 FTL 요청에 대하여, 선행 요청을 위한 재사상 작업이 시작된 후에 후행 요청이 도착하였다면, 후행 요청은 재사상 작업이 완료된 이후에만 처리가 시작되어야 한다.

실행 규칙3: 동일 의사 물리 블록으로 향하는 두 FTL 요청에 대하여, 선행 요청을 위한 재사상 작업이 시작하기 전에 후행 요청이 도착하여 처리가 시작되었다면, 후행 요청의 기존 작업은 무효화되고 재사상 작업이 완료된 이후에 재수행되어야 한다.

본 논문에서는 이러한 실행 규칙을 지키면서 요청 처리 지연을 최소화하는 병렬화된 BMS의 설계 방안을 제시하였다. 병렬화된 BMS는 상위 계층의 요청 처리 과정을 복수의 단계로 구분하고 각 단계를 독립적으로 동작하는 모듈들이 담당함으로써 요청 처리의 병렬성을 극대화한다. 또한 재사상 블록 필터와 사상 정보 캐시와 같은 자료 구조를 도입하여 BMS의 성능 병목이 될 수 있는 재사상 테이블 탐색 작업을 회피하였고, 상위 계층과의 주메모리 공유로 인한 성능 저하 문제를 방지하였다. 이와 같은 자료 구조와 동작 빈도가 높은 내부 모듈들은 복잡도가 작기 때문에 하드웨어를 통한 가속화 기법을 적용하여 성능을 더욱 개선할 수 있다.

제안한 설계 방안을 소프트웨어 및 하드웨어로 구현하였고, 임의의 동기적/비동기적 오류가 중첩하여 발생하는 시뮬레이션 환경에서 동작의 정확성을 검증하였다. 또한 무순서 플래시 메모리가 탑재된 FPGA 개발 보드 상에서 구현한 BMS의 유효성과 실효성을 평가하였다. 실험 결과, 플래시 메모리 버스가 2개 이하인 작은 용량의 저장 장치에서는 소프트웨어로 구현된 BMS도 플래시 제어기가 제공하는 대역폭을 충분히 활용할 수 있음을 확인하였다. 그러나 SSD와 같이 다수의 플래시 메모리 버스를 사용하는 대용량 저장 장치에서는 소프트웨어 개입으로 인한 요청 처리의 지연이 심각한 성능 저하를 초래할 수 있으며, 하드웨어 기반의 최적화된 BMS를 통해 성능 저하 문제를 효과적으로 극복할 수 있음을 확인하였다. FPGA에서 하드웨어 BMS를 합성한 결과, 플래시 제어기와 같은 다른 구성 요소에 비해 자원 사용량이 작으며, 이는 하드웨어 BMS가 여러 가지 응용 제품에 적용 가능함을 의미한다.

향후 과제로서 설계한 BMS의 정확성을 정형적으로 검증하는 연구가 필요하다. 제시한 설계 기법은 특정 BMS 구현에 종속되지 않으므로 다양한 BMS 구현에 적용될 수 있기 때문에, 가능한 모든 구현에 대해 실험을 통하여 정확성을 검증하는 것은 불가능하다. 또한 시뮬레이션이나 시제품 테스트 기반의 정확성 검증은 시간적 제약이라는 근본적인 문제로 인하여 완전 무결한 검증을 수행할 수 없다. 따라서 BMS의 동작을 추상화된 모델을 통하여 정형적으로 검증하는 것이 향후 중요한 연구 과제가 될 것이다.

그 밖에 BMS의 기능과 인터페이스를 확장하여 시스템 전체적인 관점에서 성능 개선에 대해 고려하는 방안도 의미 있는 연구 주제가 될 것이다. BMS는 불량 블록 처리를 위해 일정한 수의 예비 블록을 할당하

는데, 이러한 예비 블록은 상위 계층에서 접근할 수 없기 때문에 플래시 메모리의 공간 활용도가 떨어지게 된다. 이를 해결하기 위하여 의사 물리 블록 공간의 용량을 동적으로 변경하는 방식을 도입할 수 있다. BMS는 최소한의 예비 블록만을 사용하고 상위 계층에 가능한 많은 양의 의사 물리 블록을 제공한다. 추후, 예비 블록이 소진되면 의사 물리 블록의 개수를 동적으로 줄임으로써 예비 블록을 보충할 수 있다. 또한 플래시 메모리 칩 간의 부하 균등화 (Load balancing)을 통해 플래시 제어기의 효율성을 높이는 방안도 생각할 수 있다. 상위 계층의 요청들은 장기적인 관점에서 모든 칩에 균등하게 전달되지만, 단기적으로는 칩별로 작업 부하의 불균형이 발생할 수 있다. 특정 칩에 대한 요청이 플래시 제어기 내부의 버퍼 용량을 초과하여 발생한다면, 다른 칩으로의 요청은 더 이상 처리될 수 없기 때문에 성능이 저하될 것이다. BMS에서 플래시 제어기의 용량 특성을 고려하여 부하 균등화를 수행한다면 플래시 제어기의 연산 스케줄링의 효율성을 향상시킬 수 있다.

참고문헌

[1] T. Kgil and T. Mudge, "FlashCache: A NAND flash memory file cache for low power web servers," in Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2006.

[2] FUSION-IO. "Cache in ioDrive" Available at http://community.fusionio.com/forums/t/97.aspx.

[3] J. Gray, "Tape is dead, disk is tape, flash is disk, RAM locality is king," Gong Show Presentation at CIDR, 2007.

[4] L.M. Grupp, J.D. Davis, and S. Swanson, "The bleak future of NAND flash memory," in Proceedings of the 10th USENIX Conference on File and Storage Technologies, 2012.

[5] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," ACM Computing Surveys, vol. 37, no. 2, pp. 138-163, 2005.

[6] E.H. Nam, B.S.J. Kim, H. Eom, and S.L. Min, "Ozone (O3): An out-of-order flash memory controller architecture," IEEE Transactions on Computers, vol. 60, no. 5, pp. 653-666, 2011.

[7] Y.J. Seong, E.H. Nam, J.H. Yoon, H. Kim, J.-y. Choi, S. Lee, Y.H. Bae, J. Lee, Y. Cho, and S.L. Min, "Hydra: A block-mapped parallel flash memory solid-state disk architecture," IEEE

Transactions on Computers, vol. 59, no. 7, pp. 905-921, 2010.

[8] Denali. "NAND flash controller IP." Available at http://www.denali.com/en/products/databahn_flash.jsp, 2010.

[9] J.H. Yun, " X-BMS: A provably-correct bad block management scheme for flash memory based storage systems," Ph.D Dissertation, Seoul National University, 2011.

[10] Y.-H. Chang and T.-W. Kuo, "A reliable MTD design for MLC flash-memory storage systems," in Proceedings of the 10th ACM International Conference on Embedded Software, 2010.

[11] S. Yamada, T. Suzuki, E. Obi, M. Oshikiri, K. Naruke, and M. Wada, "A self-convergence erasing scheme for a simple stacked gate Flash EEPROM," in Proceedings of the Internationa Electron Devices Meeting, 1991.

[12] M. Momodomi, T. Tanaka, Y. Iwata, Y. Tanaka, H. Oodaira, Y. Itoh, R. Shirota, K. Ohuchi, and F. Masuoka, "A 4 Mb NAND EEPROM with tight programmed V(t) distribution," IEEE Journal of Solid-State Circuits, vol. 26, no. 4, pp. 492-496, 1991.

[13] J. Brewer and M. Gill, Nonvolatile Memory Technologies with Emphasis on Flash: A Comprehensive Guide to Understanding and Using Flash Memory Devices, Wiley-IEEE Press, 2011.

[14] K.-D. Suh, B.-H. Suh, Y.-H. Lim, J.-K. Kim, Y.-J. Choi, Y.-N.

Koh, S.-S. Lee, S.-C. Kwon, B.-S. Choi, and J.-S. Yum, "A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme," IEEE Journal of Solid-State Circuits, vol. 30, no. 11, pp. 1149-1156, 1995.

[15] G. Hemink, T. Tanaka, T. Endoh, S. Aritome, and R. Shirota, "Fast and accurate programming method for multi-level NAND EEPROMs," in Proceedings of the Symposium on VLSI Circuits, 1995.

[16] M. Kato, N. Miyamoto, H. Kume, A. Satoh, T. Adachi, M. Ushiyama, and K. Kimura, "Read-disturb degradation mechanism due to electron trapping in the tunnel oxide for low-voltage flash memories," in Proceedings of the International Electron Devices Meeting, 1994.

[17] R.-i. Yamada, Y. Mori, Y. Okuyama, J. Yugami, T. Nishimoto, and H. Kume, "Analysis of detrap current due to oxide traps to improve flash memory retention," in Proceedings of the 38th Annual IEEE Internaltional Reliability Physics Symposium, 2000.

[18] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Improving flash wear-leveling by proactively moving static data," IEEE Transactions on Computers, vol. 59, no. 1, pp. 53-65, 2010.

[19] D. Jung, Y.-H. Chae, H. Jo, J.-S. Kim, and J. Lee, "A group-based wear-leveling algorithm for large-capacity flash

memory storage systems," in Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, 2007.

[20] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L.R. Nevill, "Bit error rate in NAND flash memories," in Proceedings of the 46th International Reliability Physics Symposium, 2008.

[21] H. Kurata, K. Otsuga, A. Kotabe, S. Kajiyama, T. Osabe, Y. Sasago, S. Narumi, K. Tokami, S. Kamohara, and O. Tsuchiya, "The impact of random telegraph signals on the scaling of multilevel flash memories," in Proceedings of the Symposium on VLSI Circuits, 2006.

[22] C.M. Compagnoni, A. Spinelli, R. Gusmeroli, A. Lacaita, S. Beltrami, A. Ghetti, and A. Visconti, "First evidence for injection statistics accuracy limitations in NAND flash constant-current Fowler-Nordheim programming," in Proceedings of the International Electron Devices Meeting, 2007

[23] R. Micheloni, R. Ravasio, A. Marelli, E. Alice, V. Altieri, A. Bovino, L. Crippa, E. Di Martino, L. D'Onofrio, and A. Gambardella, "A 4Gb 2b/cell NAND flash memory with embedded 5b BCH ECC for 36MB/s system read throughput," in Proceedings of the IEEE Internaltion Solid-State Circuits Conference, 2006.

[24] W. Liu, J. Rho, and W. Sung, "Low-power high-throughput BCH error correction VLSI design for multi-level cell NAND flash memories," in Proceedings of the IEEE Workshop on Signal Processing Systems Design and Implementation, 2006.

[25] R. Micheloni, R. Ravasio, A. Marelli, E. Alice, V. Altieri, A. Bovino, L. Crippa, E. Di Martino, L. D'Onofrio, and A. Gambardella, "A 4Gb 2b/cell NAND flash memory with embedded 5b BCH ECC for 36MB/s system read throughput," in Proceedings of the IEEE International Solid-State Circuits Conference, 2006.

[26] Y. Maeda and H. Kaneko, "Error control coding for multilevel cell flash memories using nonbinary low-density parity-check codes," in Proceedings of the IEEE Interntional Symposium on Defect and Fault Tolerance in VLSI Systems, 2009.

[27] G. Dong, N. Xie, and T. Zhang, "On the use of soft-decision error-correction codes in NAND flash memory," IEEE Transations on Circuits and Systems, vol. 58, no. 2, pp. 429-439, 2011.

[28] S. Aritome, R. Shirota, G. Hemink, T. Endoh, and F. Masuoka, "Reliability issues of flash memory cells," Proceedings of the IEEE, vol. 81, no. 5, pp. 776-788, 1993.

[29] A. Nozoe, H. Kotani, T. Tsujikawa, K. Yoshida, K. Furusawa, M.

Kato, T. Nishimoto, H. Kume, H. Kurata, and N. Miyamoto, "A 256-Mb multilevel flash memory with 2-MB/s program rate for mass storage applications," IEEE Journal of Solid-State Circuits, vol. 34, no. 11, pp. 1544-1550, 1999.

[30] J.-D. Lee, S.-H. Hur, and J.-D. Choi, "Effects of floating-gate interference on NAND flash memory cell operation," IEEE Electron Device Letters, vol. 23, no. 5, pp. 264-266, 2002.

[31] T.-S. Jung, Y.-J. Choi, K.-D. Suh, B.-H. Suh, J.-K. Kim, Y.-H. Lim, Y.-N. Koh, J.-W. Park, K.-J. Lee, and J.-H. Park, "A 117-mm2 3.3-v only 128-mb multilevel NAND flash memory for mass storage applications," IEEE Journal of Solid-State Circuits, vol. 31, no. 11, pp. 1575-1583, 1996.

[32] J. De Blauwe, J. van Heudt, D. Wellekens, G. Groeseneken, and H.E. Maes, "SILC-related effects in flash E2PROM's-Part I: A quantitative model for steady-state SILC " IEEE Transations on Electron Devices, vol. 45, no. 8, pp. 1745-1750, 1998.

[33] H.P. Belgal, N. Righos, I. Kalastirsky, J.J. Peterson, R. Shiner, and N. Mielke, "A new reliability model for post-cycling charge retention of flash memories," in Proceedings of the 40th Annual Reliability Physics Symposium, 2002.

[34] R.-i. Yamada, T. Sekiguchi, Y. Okuyama, J. Yugami, and H. Kume, "A novel analysis method of threshold voltage shift due

to detrap in a multi-level flash memory," in Proceedings of the IEEE Symposium on VLSI Technology, 2001.

[35] L.M. Grupp, A.M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf, "Characterizing flash memory: anomalies, observations, and applications," in Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009.

[36] H.-W. Tseng., L. Grupp., and S. Swanson., "Understanding the impact of power loss on flash memory," in Proceedings of the 48th Design Automation Conference (DAC), 2011.

[37] ONFI. "ONFI 3.1 Specification." Available at http://onfi.org/specifications/, 2012.

[38] K.J. Lee, Y.J. Song, T.J. Cho, S.H. Chang, C.C. Lee, B.S. Song, and J.H. Choi, "Multi-chip package," US Patent, no. 6,087,722, 2000.

[39] "MTD manual page." Available at http://www.linux-mtd.infradead.org/index.html.

[40] "UBI manual page." Available at http://www.linux-mtd.infradead.org/doc/ubi.html.

[41] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A design for high-performance flash disks," ACM SIGOPS Operating System

Review, vol. 41, no. 2, pp. 88-93, 2007.

[42] M. Wu and W. Zwaenepoel, "eNVy: a non-volatile, main memory storage system," in Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[43] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in Proceedings of the USENIX Annual Technical Conference, 1995.

[44] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," IEEE Transactions on Consumer Electronics, vol. 48, no. 2, pp. 366-375, 2002.

[45] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," ACM Transactions on Embedded Computing Systems, vol. 6, no. 3, 2007.

[46] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim, "A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications," ACM Transactions on Embedded Computer Systems, vol. 7, no. 4, pp. 1-23, 2008.

[47] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level

address mappings," in Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, 2009.

[48] Y.J. Seong, "Formal verification of a compositional FTL design framework," Ph.D Dissertation, Seoul National University, 2013.

[49] M. Rosenblum and J.K. Ousterhout, "The design and implementation of a log-structured file system," ACM Transactions on Computer Systems, vol. 10, no. 1, pp. 26-52, 1992.

[50] M.-L. Chiang, P.C.H. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," Software Practice & Experience, vol. 29, no. 3, pp. 267-290, 1999.

[51] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage," in Proceedings of the the 6th USENIX Conference on File and Storage Technologies, 2008.

[52] J.H. Yoon, E.H. Nam, Y.J. Seong, H. Kim, B.S. Kim, S.L. Min, and Y. Cho, "Chameleon: A high performance flash/FRAM hybrid solid state disk architecture," IEEE Computer Architecture Letters, vol. 7, pp. 17-20, 2008.

[53] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based

flash translation layer for NAND flash memory," in Proceedings of the 6th ACM & IEEE International Conference on Embedded Software, 2006.

[54] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-aware sector translation for NAND flash memory-based storage systems," ACM SIGOPS Operating System Review, vol. 42, no. 6, pp. 36-42, 2008.

[55] J.H. Yun, J.H. Yoon, E.H. Nam, and S.L. Min, "An abstract fault model for NAND flash memory," IEEE Embedded Systems Letters, 2012.

[56] Micron Inc., "Bad block management in NAND flash memory," Technical Note-29-59, 2011.

[57] A.M. Caulfield, L.M. Grupp, and S. Swanson, "Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications," in Proceedings of the Architectural Support for Programming Languages and Operating Systems, 2009.

[58] L.-P. Chang and T.W. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," in Proceedings of the 8th Real-Time and Embedded Technology and Applications Symposium, USA, 2002.

[59] S.-K. Jo, "Flash memory device for performing bad block

management and method of performing bad block management of flash memory device," US Patent, no. 7,434,122 B2, 2008.

[60] J.-i.S. Wook-ghee Hahn, Youn-yeol Lee, "Non-volatile memory device and bad block remapping method," US Patent, no. 8,050,093 B2, 2011.

[61] C.-K.Y. Chien-Hua Chu, Horng-Sheng Yan, "Block management and replacement method, flash memory storage system and controller using the same," US Patent, no. 8,074,128 B2, 2011.

[62] 김진혁, 이양섭, 김영곤, "메모리 장치의 배드 블록 주소를 번역하는 방법, 메모리장치의 배드 블록 주소를 번역하는 장치 및 이를 포함하는메모리 장치 컨트롤러," 대한민국 특허, no. 10-2008-0030805, 2008.

[63] C. Wang and W.-F. Wong, "Extending the lifetime of NAND flash memory by salvaging bad blocks," in Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, 2012.

[64] NVM express. "NVM express 1.1 specification." Available at http://www.nvmexpress.org/, 2012.

[65] F. Schmidt, The SCSI Bus and IDE Interface: Protocols, Applications and Programming, Addison-Wesley Longman Publishing Co., Inc., 1997.

[66] K. Smith, "Understanding SSD over provisioning," in Flash Memory Summit, 2012.

[67] 강운학, 이상원, "카운팅 블룸필터를 활용한 FAST FTL 로그영역 탐색성능 개선," 정보과학회논문지: 데이타베이스, vol. 38, no. 3, pp. 149-164, 2011.

[68] 김기준, 남이현, 윤진혁, "NAND 플래시메모리 특성모델 기반 시뮬레이션 프레임워크," 한국정보과학회 학술발표논문집, vol. 39, no. 2A, pp. 93-95, 2012.

[69] Xilinx. "Virtex-5 FPGA family." Available at http://www.xilinx.com/products/virtex5/index.htm, 2010.

Abstract

Performance optimization of a bad block management scheme with guaranteed correctness in parallelized flash memory storage devices

Hongseok Kim

Department of Electrical Engineering and Computer Sciences

The Graduate School

Seoul National University

In a flash memory storage device where multiple flash memory operations are executed in parallel, if faults are frequent while flash operations are executing it is difficult to guarantee the correctness of the flash memory management software such as FTL because of the dependences between flash memory operations. Furthermore in a practical flash memory storage device, minimizing bad block management overhead from flash memory faults is critical for optimizing performance.

This dissertation presents a bad block management scheme that resolves the dependence problem among flash memory operations associated with the bad block handling process in an environment where multiple flash memory operations may be outstanding. The presented scheme not only guarantees the correctness of bad block management, but also allows normal flash memory operations while han-

dling bad blocks to prevent delayed handling of multiple operations generated by upper layer flash memory management software. Also, address translation overhead introduced while handling the flash operations from FTL is accelerated by utilizing bloom filter and map data caching to optimize the performance of the bad block management scheme. The presented bad block management scheme can be implemented either in hardware or software depending on the performance requirement of the flash memory storage device and available hardware resource.

We implement the bad block management scheme in both software and hardware, and verify its correctness in a simulation environment where arbitrary nested synchronous and asynchronous faults may occur while executing multiple flash memory operations. In order to demonstrate the validity and practicality of the bad block management scheme, we also evaluate its performance on an FPGA development board with a flash memory controller exploiting multi-chip parallelism. Experimental results show that in a storage device with small number of flash memory chips software only implementation utilizes over 95% of the bandwidth provided by the flash memory controller. However, in a high performance storage device such as SSD that uses many flash memory chips, software only approach leads to performance degradation due to excessive software intervention. On the other hand, hardware-assisted implementation for high performance storage device utilizes over 97% of the flash memory controller bandwidth.

Key words: flash memory, bad block, bad block management, flash translation layer (FTL)

***Student Number*:** 2005-21375