낸드플래시 구현 프로그램 계획서

과 목	스토리지시스템
담 당 교 수	권 세 진
학 번	201720970
학 과	소프트웨어·미디어·산업공학부
이 름	권 대 한

목 차

1. 요구분석

1-1) 낸드플래시 구현 프로그램이란?

NAND Flash의 성능을 측정하기 위해, 프로토타입과 같은 물리 메모리에 직접 데이터를 쓰는 대신 NAND Flash의 특성을 프로그램으로 구현하여, NAND Flash에서 작동한 것과 같은 결과를 얻게 해주는 프로그램이다.

FTL은 SSD 내부에 존재하고 있으며, SATA, PCI-e와 같은 Interface를 통해 Host Interface로 데이터 요청이 들어왔을 때, 매핑 테이블을 참조해서 해당 요청을 중간 단계에서 번역해주는 역할을 하고 있습니다. 그리고 HDD 기반의 File System을 주로 사용하는 현 Software 생태계에서 FTL은 SSD에게 없어서 안 될 존재가 되었다.

그러므로 결국 FTL 기법마다 데이터를 쓰는 조건, 데이터를 지우는 조건이 달라지므로, SSD 친화적으로 데이터 요청을 받아주는 FTL의 개선은 꾸준히 진행되어야 한다.

그래서 우리 프로그램은 테스트 시 추가로 측정된 Write/Erase 횟수를 기반으로, FTL 기법 간 우월 정도를 측정하는 것을 중점으로 만들어진 프로그램을 만들었다.

우리가 이 프로그램을 고안할 때 생각했던 문제점은 NAND Flash에서 한 번 쓰인 위치를 다시 쓰고자 할 때, write-before-erase의 특성으로 인해, 무조건 비워줘야 하는 특성이 존재한다는 것이다. 그리고 여기서 큰 고려사항 2가지가 발생한다.

첫 번째 문제는 NAND Flash 특성상 지울 수 있는 횟수가 제한되어 있다는 것이며,

두 번째 문제는 NAND Flash의 실질적인 read/write 단위와 erase의 단위가 다르다는 점에서 어떤 방법으로 해결하는지에 따라서 사용자가 SSD를 사용할 때 실 성능에 큰 영향을 주므로, 우리가 만든 프로그램에서는 정적/동적 섹터, 블록 매핑의 추가 쓰기 횟수, 지우기 횟수를 비교할 수 있게 제작하였으며, 이를 통해, 프로그램 사용자는 본인이 입력한 데이터 패턴에서 어떤 기법이 제일 유리한지 바로 파악할 수 있게 된다.

본래 FTL 기법을 테스트하기 위해서는 프로토타입 보드에 테스트하고자 하는 NAND Flash와 FTL 기법을 설치해서 작동환경을 구성해줘야 한다.

이후 사용자가 주로 사용하는 애플리케이션의 데이터 패턴을 참조해서 실물의 NAND Flash에 쓰도록 하는 것으로 테스트가 진행되지만, 프로토타입으로 진행한다면, 시간이 상당히 많이 소비된다는 문제점이 발생한다. 추가로 테스트를 돌리면서 실물의 NAND Flash가 마모되는 문제도 발생하므로, 마모도에 구애받지 않는 NAND의 특성을 기반한 프로그램은 꼭 필요하다.

단지 프로그램으로 FTL 기법을 테스트함으로, NAND Flash의 성능을 크게 개선 시킬 수 있다는 것에 큰 의의가 있다.

이 프로그램으로 FTL 기법을 테스트한다면, 소요 시간, 비용 모두 줄일 수 있으며, 아주 효율적으로 FTL 기법을 개선 시킬 수 있을 것이다.

1-2) 정적/동적 매핑 기법 소개

데이터 영역를 매핑하는 기법에는 크게 두 가지가 존재한다.
데이터 read/write 단위로 매핑하는 Sector Mapping 기법, 두 번째로 데이터 erase 단위로 매핑하는 Block Mapping 기법으로 이와 같은 기법으로 두 가지를 사용한다.

먼저 Sector Mapping의 특징으로 데이터 read/write 단위로 Mapping 되는 기법이므로 NAND Flash 가용 용량이 늘어날수록 상당하게 큰 Mapping Table을 유지해야 한다는 점에서 데이터를 읽어올 때 결국 FTL 내 Mapping Table의 매핑 정보 기반으로 데이터 접근이 가능하므로, 데이터의 빠른 접근을 위해서 SRAM, DRAM등의 상당히 빠르고 비싼 메모리에 Mapping Table을 유지해야 합니다. 그러나 비어있는 모든 위치에 데이터를 쓸 수 있다는 것이 큰 장점이며, 데이터 영역의 공간이 꽉차게 되어 지우기 연산이 발생하였다면 연관된 데이터 블록(영역)을 모두 비울 필요가 없다는 것이 Sector Mapping 기법의 작동 방식이다.

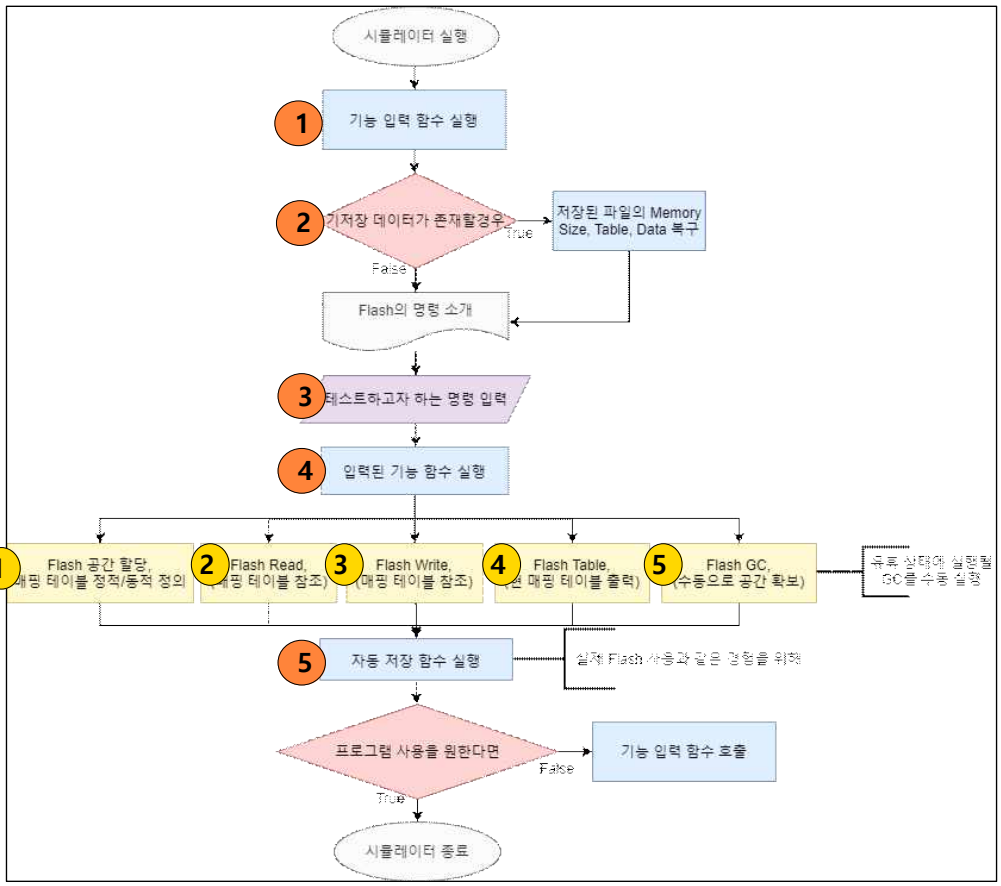
Block Mapping의 경우 데이터 erase 단위로 Mapping 되는 기법이므로, Sector Mapping과 비교해서 NAND Flash의 가용 용량이 늘어나도 DRAM, SRAM에 유지해야 하는 Mapping Table의 용량은 커지긴 하지만, 1개의 Block에 32개의 Sector를 가지고 있다면, 유지해야 되는 데이터 크기는 Sec_Map_Data_Size /= 32의 크기를 가지게 되므로, 생산비용의 측면에서 아주 유리한 Mapping 기법이라고 볼 수 있다.
그러나 Block Mapping의 경우, Offset(LPN % numofblk) 기반 데이터 접근밖에 못 하므로, 목적지 블록에 다른 오프셋에 빈 공간이 있더라도, Sector Mapping과 다르게 데이터 저장이 불가하다는 특징을 가지고 있다. 추가로 하나의 데이터 블록에서 Overwrite에 대한 모든 데이터를 받았다면, 기법에 따라 다르지만 1~32개의 블록을 동시에 지워서 유효하지 않은 데이터를 정리해줘야 하는 경우도 생기게 되지만, 최선의 대안으로 정리가 필요할 때마다 해당 블록에 대해서만 처리한다면, 1개의 데이터 블록을 비움으로, 1개의 데이터를 얻을 수 있게 된다. 그러나 이 경우에는 이미 데이터를 받을 만큼 받은 상태이므로, write 요청이 들어올 때마다 꼭 블록을 1개씩 비워줘야 하는 문제가 발생된다. (Lazy) 그리고 고르게 마모시키는 웨어 레벨링의 고려에 따라 성능이 크게 차이 날 수 있다고 생각된다.

이러한 매핑 기법이 존재하지만, 방금 설명한 내용들은 Static(정적 할당)에서의 가정이라 볼 수 있다. Mapping Table의 구성 방식을 정적/동적으로 구분할 수 있는데, 간단히 설명해서 데이터 할당 전에 Mapping 정보를 사전에 구성해두는 것인지(정적), 그리고 데이터 할당 마다 빈 공간을 색인해서, 데이터가 Mapping Table을 참조해서 저장 하는 방식으로 구분된다.

먼저 Static(정적)의 경우, 이미 데이터 공간에 대한 Sector, Block이 이미 설정되어 있으므로, 추가적인 공간을 꼭 할당해서 Overwrite과 같은 부가 기능을 구현해야 하지만 Dynamic(동적)의 경우, 설계를 어떻게 했는지에 따라, Static(정적)과 비교해서 사용자에게 추가 공간을 할당시킬 수 있다. 유효한 페이지 복사 공간을 최소화하는 것이다. 이 경우에는 사용 중인 블록의 크기가 어느 정도일 때 무조건 Garbage Collection을 실시함으로 Delay가 발생되지만, 사용자에게 보장해줄 수 있는 저장 공간을 극대화할 것이다.

1-3) 전체적인 프로그램 구성 계획

<프로그램 구성도>



(1) Dynamic/Static, Sector/Block Mapping 기법에서 데이터 할당 방식이 조금 다르므로 초기 프로그램 실행 시 사용자에게 사용할 매핑 기법을 선택받는다.

(2) 기존에 작업하던 내용이 존재한다면, 마지막 수행 작업 기준의 저장파일을 불러와 Flash의 공간을 할당 후, Mapping 정보, 데이터를 복구하며 사용자에게 명령을 소개한다.

(3) 데이터가 존재하지 않는다면, 명령을 소개하면서 사용자의 입력을 대기한다.

(4-1) 사용자에게 입력받은 명령을 실행하며, 할당 명령이 들어왔다면, 입력받은 크기로 메모리를 할당한다. 이후 사용자에게 입력된 할당 기법으로 매핑 테이블 공간, 정보를 할당한다.

(4-2) 읽기 명령이 실행된 경우 매핑 테이블을 참조해서 해당 섹터의 데이터를 출력시켜주는 역할을 한다.

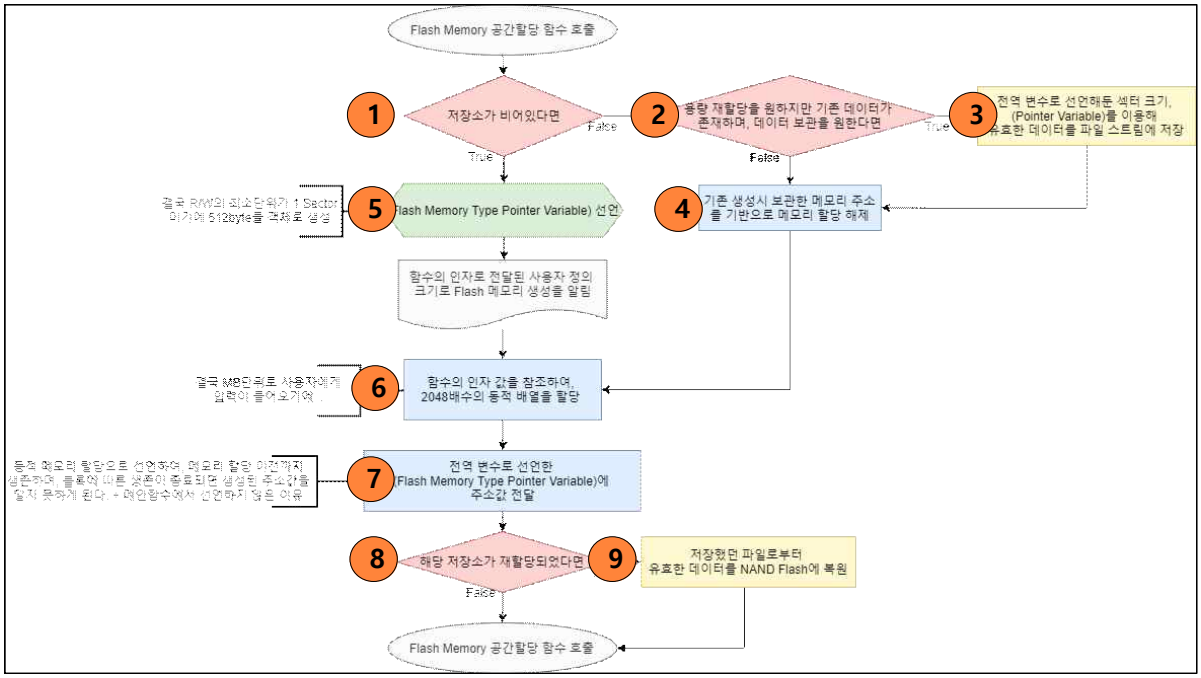
(4-3) 정적 할당의 쓰기 명령은 매핑 테이블을 참조해서 요청 데이터를 쓰게 되며, 동적 할당의 경우, 매핑 테이블 할당을 위해, 빈 공간을 색인 후, 데이터를 쓰게 될 것이다. 여기서 Overwrite가 발생된 경우 내부적으로 추가 처리를 하게 된다.

(4-4) Table 출력 명령의 경우 현 매핑 테이블의 데이터를 모두 출력한다.

(4-5) 마지막으로 GC 명령의 경우 유휴시간에 자동으로 수행되지만, 명령 실행 시 수동으로 유효한 데이터를 남겨두고 모든 유효하지 않은 데이터를 정리해주는 명령이다.

(5) 이후 각 기능을 마쳤다면, 자동 저장 함수를 호출하여 파일 타입으로 현재까지 테스트 내용을 저장한다.

<FTL Flash 할당>



(1) 저장소가 할당된 적이 없거나, 파일 입출력 로드에 실패하였다면 저장소가 비어있게 되는데, 초기에 이를 확인한다.

(2) 저장소가 할당되어 있으며, 기존의 데이터가 존재할 경우, 입력한 크기로 재할당을 할 것인지 선택받는다.

(3) 전역변수의 저장소 시작 주소를 기반으로 섹터의 여유 공간을 참조하여 저장소의 모든 데이터를 파일 형태로 임시 보관한다.

(4) 동적 객체 배열 Type으로 구성되었으므로 재할당을 위한 전역 포인터 변수를 이용해 메모리 해제를 실시한다.

(5) 저장소 할당을 위해 Flash Type의 포인터를 선언한다.

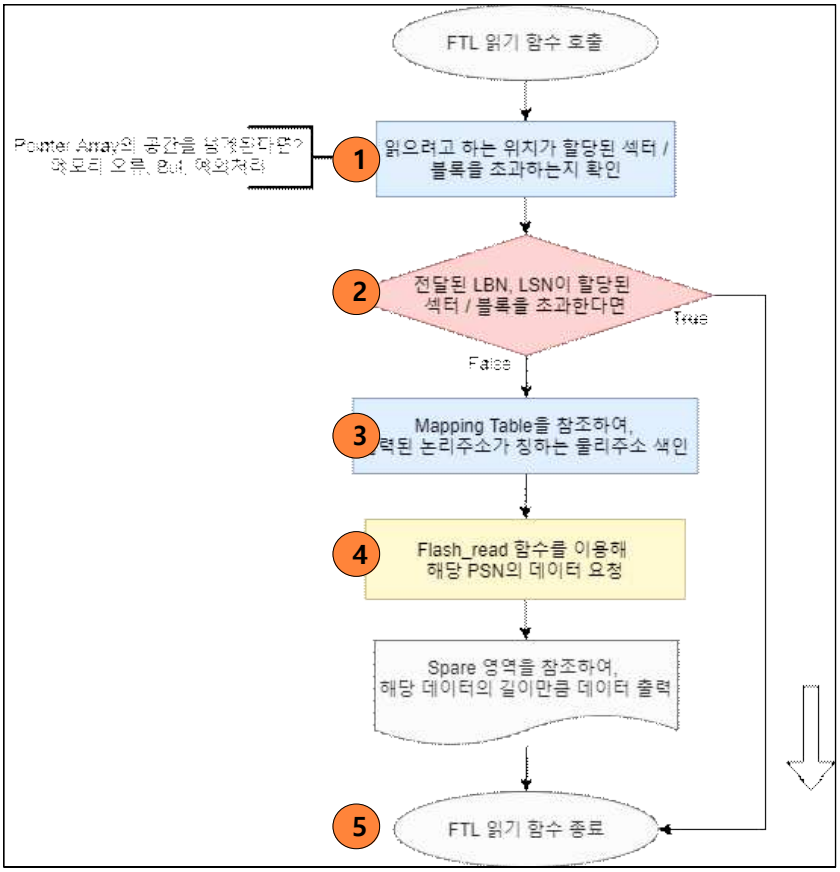
(6) 1MB는 1048576Byte, 2048개의 섹터로 구성되어 있으므로 입력 값의 2048배수 섹터를 할당한다.

(7) 할당 함수가 종료되어도 추후 다른 함수에서 해당 주소에 접근할 수 있도록 전역 포인터 변수에 주소를 저장해준다.

(8) 처음 실행시 Trigger된 재할당을 작동시켰을 경우를 확인한다.

(9) 만약 재할당을 하였다면, 임시 저장한 파일로부터 매핑 정보, Flash 데이터를 복구한다.

<FTL 읽기 함수>



(1) 현재 읽고자 하는 섹터가 전역변수로 선언한 섹터 개수(현 섹터 개수)의 크기를 넘는지 확인한다.

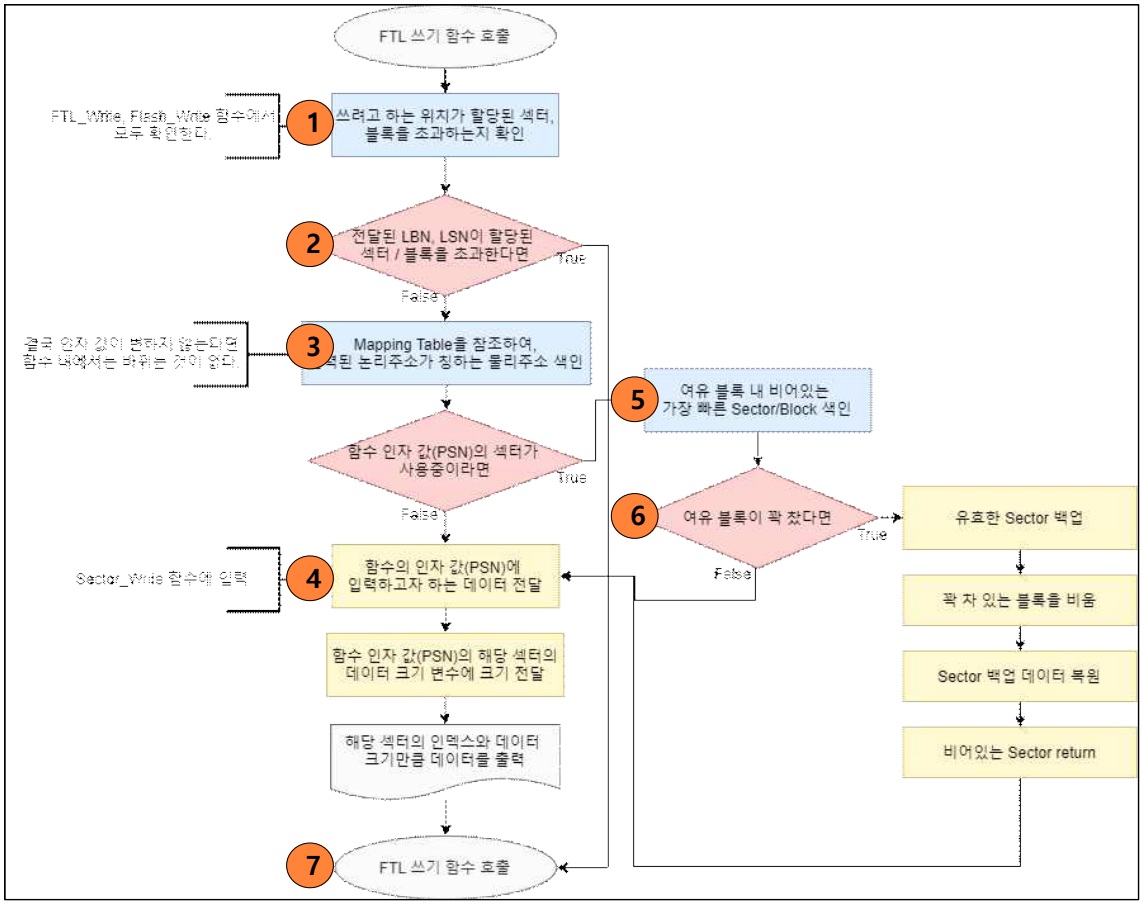
(2) 입력된 LSN이 할당 섹터 크기보다 크다면 오류 메시지와 함께 함수를 종료한다. 특히, Spare 영역을 읽고자 할 때 이를 차단한다.

(3) 할당된 섹터를 초과하지 않는 엑세스의 경우, 매핑 테이블을 참조해서, 입력된 LSN의 실제 위치인 PSN을 찾아낸다.

(4) 해당 PSN을 실제 Flash 읽기 함수에 인자 값으로 전달해줌으로 실제 데이터를 출력한다.

(5) 모든 작업이 끝나면 읽기 함수를 종료한다.

<FTL 쓰기 함수>



(1) 현재 쓰고자 하는 섹터가 전역변수로 선언한 섹터 개수(현 섹터 개수)의 크기를 넘는지 확인한다.

(2) 입력된 LSN이 할당 섹터 크기보다 크다면 오류 메시지와 함께 함수를 종료한다. 특히, Spare 영역을 읽고자 할 때 이를 차단한다.

(3) 할당된 섹터를 초과하지 않는 엑세스의 경우, 매핑 테이블을 참조해서, 입력된 LSN의 실제 위치인 PSN을 찾아낸다.

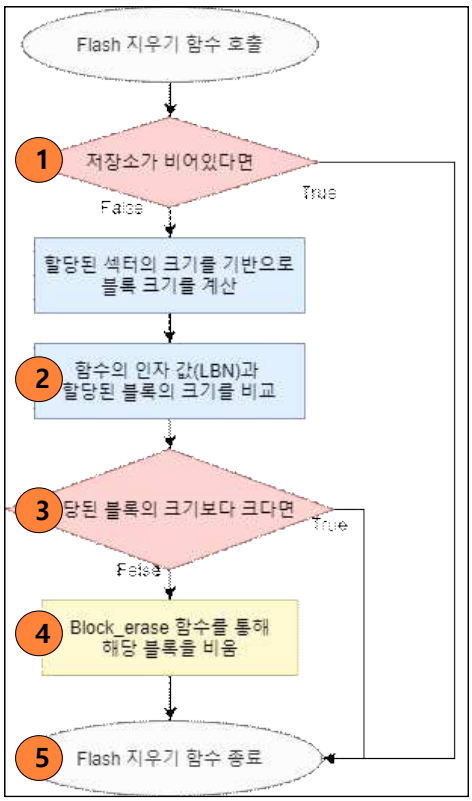
(4) 현재 요청된 섹터가 비어있어 데이터를 쓸 수 있다면, 실질적으로 데이터를 쓰는 함수인 Sector_write(Flash_write)에 요청된 데이터와 데이터 크기를 전달한다.

(5) 반대로 현재 요청된 위치에 데이터를 쓸 수 없다면, 요청된 할당에서 추가로 쓸 수 있는 가장 빠른 위치를 찾아낸다.

(6) 5번에서 데이터를 쓸 위치를 찾아내지 못했다면, 데이터 블록, 여유 블록의 유효하지 않은 데이터를 지워주기 위해서, 별로도 할당되지 않은 공간 혹은 RAM에 유효 데이터를 백업함으로 데이터 무결성을 지켜준다. 이후 꽉 찬, 방금 백업한 블록을 비워주고, 모든 블록을 스캔하고 이와 같은 작업을 반복해서 데이터 공간을 확보하며, 이후 요청된 쓰기를 처리한다.

(7) 모든 작업이 끝나면 쓰기 함수를 종료한다.

<FTL 지우기 기능>



(1) 지우기 명령이 입력되었다면, FTL_erase 함수에서, 전역 포인터 변수를 통해, 저장소가 할당되어있는지 확인한다.

(2) 지우고자 하는 블록이 할당 섹터 크기보다 크다면 오류 메시지와 함께 함수를 종료한다.

(3) 할당된 블록을 초과하는 지우기 명령을 확인하고, 이를 넘는다면, FTL_erase 함수를 종료한다.

특히, Spare 영역을 지우고자 할 때 차단한다.

(4) 그 반대의 경우, 초과하지 않는 경우 Block_erase(Flash_erase) 함수를 통해 해당 블록을 지우게 된다. (GC를 위한 함수가 아니므로)

(5) 모든 작업이 끝나면 지우기 함수를 종료한다.

2. 매핑 기법의 논리 구성도

2-1) 각 FTL 기법의 고려사항

Case 1) 정적 섹터 매핑 방식

(1) 해당 공간에 처음 쓰기 요청이 들어온 경우

정적 할당의 경우, 데이터 쓰기 요청이 들어오기 전에 이미 매핑 정보가 사전 결정되어 있으므로, 해당 매핑 정보에 맞춰서 데이터를 저장하게 된다.

(2) 해당 공간에 이미 데이터가 들어 있는 경우

이미 데이터가 할당되어 있다면 NAND Flash의 특성에 따라 비워진 곳만 Write가 가능하다. 이에 대한 대안으로 사용자에게 보이지 않는 여분의 공간에 요청 데이터를 저장함으로, 추가 연산에 대한 발생을 지연시킬 수 있다.

(3) 여유 블록만 꽉 찬 경우

(2)번과 같은 이유로 Overwrite이 많이 발생해서 여유 블록을 모두 채우게 되었다면, 다음 요청이 도달하였을 때, 꽉 찬 여유 블록의 유효한 데이터만 비어있는 여분 블록에 저장시킬 것이며, 마지막으로 매핑 정보를 수정해 줄 것이다.

여유 블록에 대한 램 Buffer를 고민해보았지만, 결국 Write 요청에서 사용하는 것이 더 좋은 효율을 보여줄 것으로 예상되므로 이와 같은 구성을 하였다.

(4) 여유 블록과 데이터 블록이 모두 꽉 찬 경우

이와 같은 경우 램 버퍼, 데이터 블록, 여유 블록(Overwrite를 위한)이 모두 꽉 찬 경우를 말하는데, 여유 블록의 유효 데이터를 기반으로, 본래 데이터 블록의 유효 데이터만 병합 전용 여유 블록에 복사하며 매핑 정보를 바꿔준다. 추가로 마지막으로 병합 전용 블록의 위치(PBN)를 기록해줘야 한다. 여기서 모든 여유 블록의 데이터가 본래 데이터 블록으로 복구되었다면 여유 블록을 비워준다.

<----- 다음 장 ----->

Case 2) 동적 섹터 매핑 방식

(1) 처음 쓰기 요청이 들어온 경우

처음 Write 요청이 들어왔다면, 매핑 테이블에 할당된 매핑 정보가 없을 것이다. 후속 쓰기 요청을 위해 마지막 Write한 PSN을 기록하는 변수를 0으로 선언한다. 쓰여진 첫 번째 섹터를 매핑 정보로 할당하며, 데이터를 저장한다.

(2) 해당 공간에 이미 데이터가 들어 있는 경우

이미 매핑된 섹터가 존재한다는 것은 데이터를 이미 할당받았다는 것과 같은 의미이므로, 마지막 Write PSN을 기록한 변수를 참조해서, 새로운 PSN에 데이터를 쓰게 된다. 이후 매핑 정보를 수정해준다.

(3) 여유 블록만 비어있을 때

동적 섹터 매핑에서 결국 여유 블록이 필요없다고 생각했지만, 순차적으로 유효한 데이터가 데이터 영역을 모두 채웠을 때, 후속 요청을 위해 여러 개의 여유공간이 필요하다.
이와 같은 이유로 여유 블록 0번까지(병합 전용 블록 전까지) 데이터를 받는다. 데이터가 꽉 찼다면, 처음 데이터 블록부터 필요 없는 데이터를 모두 비워준다.

(4) 여유 블록과 데이터 블록이 모두 꽉 찬 경우

이와 같은 경우 데이터 블록, 여유 블록(Overwrite를 위한)이 모두 꽉 찬 경우를 말하는데, 여유 블록의 유효 데이터를 기반으로, 본래 데이터 블록의 유효 데이터만 병합 전용 여유 블록에 복사하며 매핑 정보를 바꿔준다. 추가로 마지막으로 병합 전용 블록의 위치(PBN)를 기록해줘야 한다. 여기서 모든 여유 블록의 데이터가 본래 데이터 블록으로 복구되었다면 여유 블록을 비워준다.

Case 3) 정적 블록 매핑 방식

(1) 해당 공간에 처음 쓰기 요청이 들어온 경우

요청 데이터는 논리 섹터 번호로 들어올 것이므로, 블록 매핑 기법에 맞도록 논리 블록 번호와 offset 계산을 해야 한다. 이후 해당 offset에 데이터를 써줌으로 작업을 마친다.

(2) 해당 공간에 이미 데이터가 들어 있는 경우

결국 블록 레벨의 매핑을 사용한다는 것은, offset 기반으로 실제 데이터 위치를 찾아낸다는 의미이므로, 기존의 데이터 블록에 이미 데이터가 쓰인 경우, 여유 블록의 offset에 데이터를 저장하게 되며, 기존 데이터 블록의 유효한 데이터가 존재한다면 여유 블록으로 옮겨온다. Overwrite시 여유 블록 0, 1의 PBN(원본 데이터 블록의 PBN)을 기록해둔다.

(3) 여유 블록의 해당 offset이 사용 중인 경우

여유 블록에 요청된 데이터를 쓰기 위해, 여유 블록의 유효한 데이터와 상응되는 데이터 블록의 유효 데이터를 두 번째 여유 블록에 복사한다. 만약, 두 개의 여유 블록이 모두 사용 중이라면, 모든 유효 하지 않은 블록을 비우며, 중복 할당을 막기 위해, Overwrite시 기록한 여유 블록 0, 1에 대한 PBN에 Overwrite 받는다. 이후 사용한 PBN 변수를 기존의 데이터 블록으로 칭해주며, 매핑 정보를 현재 데이터 블록으로 수정해준다.

(4) 여유 블록과 데이터 블록이 모두 꽉 찬 경우

이 경우에 여유 블록에 상응되는 데이터 블록에서 Overwrite이 발생된 것이므로, 유효하지 않은 데이터가 존재하는 모든 블록을 순차적으로 비워준다.

Case 4) 동적 블록 매핑 방식

(1) 해당 공간에 처음 쓰기 요청이 들어온 경우

요청 데이터는 논리 섹터 번호로 들어올 것이므로, 블록 매핑 기법에 맞도록 논리 블록 번호와 offset 계산을 해주고, 비어있는 데이터 블록를 할당받아 저장한다. 여기서 동적 섹터 매핑과 같이 Write한 PBN 번호 변수를 선언하여 기록한다.

(2) 해당 공간에 이미 데이터가 들어 있는 경우

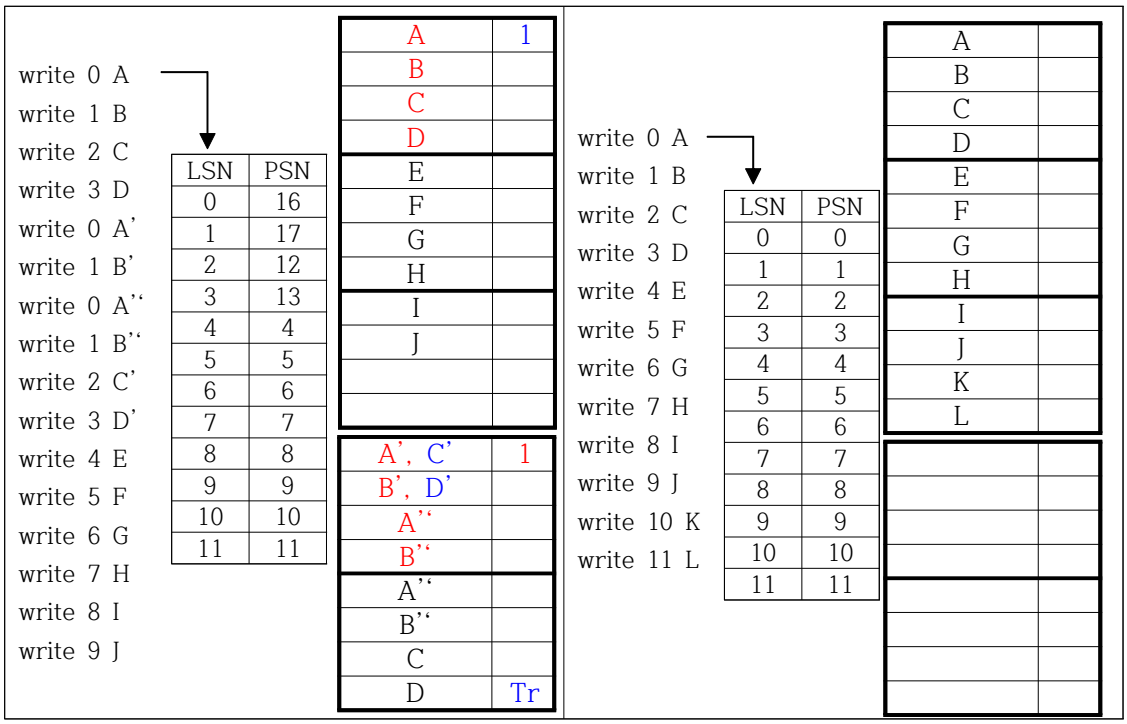
이미 해당 LBN에 대한 데이터 블록이 할당되어 있으며, 해당 offset에 데이터가 들어 있을 때, 웨어 레벨링을 고려하여, 요청된 데이터와 기존의 유효 데이터를 저장하기 위해, (1)번에서 기록한 PBN 번호의 다음 데이터 블록에 데이터를 쓰도록 한다. 여기서 사용자에게 사용공간을 보장하기 위해서 최소 1개의 데이터 블록을 확보해둬야 한다.

(3) 여유 블록과 데이터 블록이 모두 꽉 찬 경우

이 경우에 여유 블록에 상응되는 데이터 블록에서 Overwrite이 발생된 것이므로, 유효하지 않은 데이터가 존재하는 모든 블록을 순차적으로 비워준다.

2-2) 정적/동적 섹터 매핑 기법

<정적 섹터 매핑 기법의 작동>

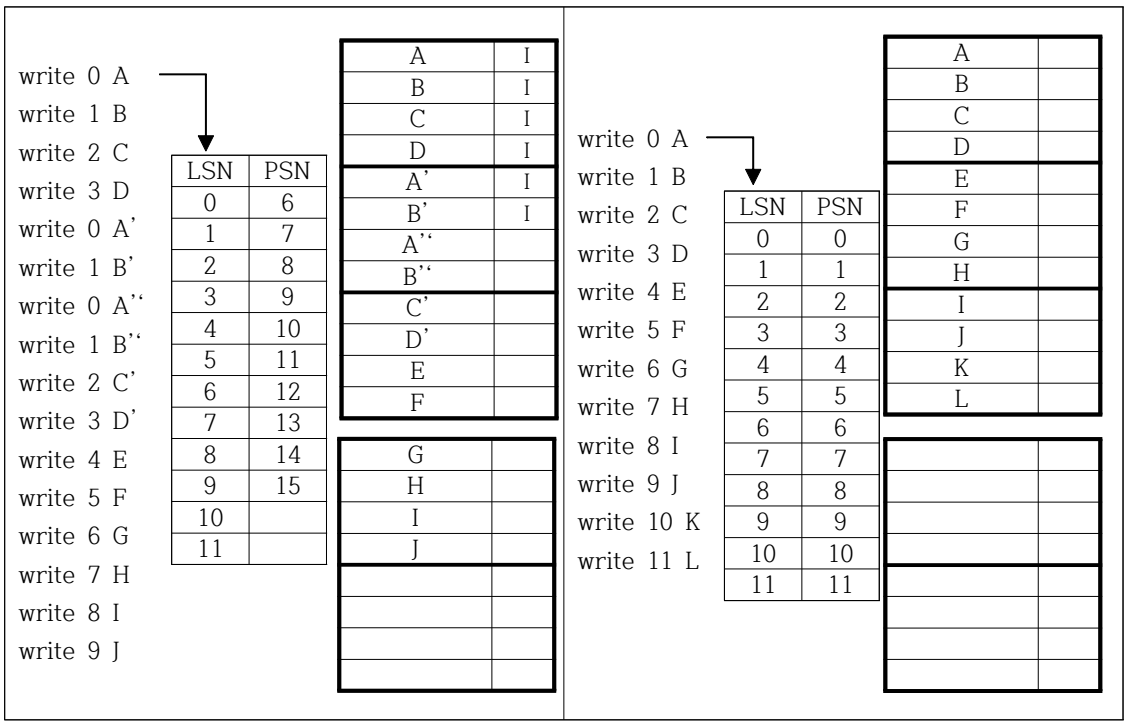


왼쪽의 경우 16개의 데이터 쓰기 요청에 대한 정적 섹터 매핑의 결과이며,
오른쪽은 12개의 순차 데이터 요청에 대한 결과이다.
3개의 데이터 블록, 2개의 여유 공간을 사용하기로 결정하였다.

Case 1) 정적 섹터 매핑의 경우 데이터 저장 위치가 사전 결정되어 있으므로, 0~3에 대한 쓰기 요청이 순차적으로 NAND에 쓰여질 것이다. 이후 A', B', A'', B''에 대한 Overwrite 요청이 여유 블록 0번에 모두 받도록 한다. 그러나 현재 DRAM을 사용하지 않으므로 후속요청 C'과 D'를 처리할 때, 해당 위치에 이미 데이터가 들어 있으며, 여유 블록 0번조차 모두 꽉 차 있으므로, 유효한 데이터만 모아서 여유 블록 1번에 저장하게 된다. 여유 블록 1번이 데이터 블록 0번으로 승격되는 것 같지만, 단지 비어있는 블록에 데이터 블록 0번, 여유 블록 0번에 대한 GC 처리 한 것이다. 여유 블록의 모든 데이터가 유효하다면 다음 여유 블록에 요청을 받으며, 다음 여유 블록이 마지막 블록이라면, 여유 블록을 비워주는 전체 GC를 실시한다. 그리고 여기서 병합 전용 블록이 데이터 블록 0번이 된다. (추후 병합작업)
이후 C', D'을 여유 블록 0번에 저장시켜주고, 각 매핑 정보를 수정해주며, 마지막으로 4~9번에 대한 쓰기 요청을 처리한다.

Case 2) 오른쪽의 경우 0~11번까지 순차적인 데이터가 들어왔으므로, 절반 이상의 유효한 데이터가 블록을 모두 채웠다. 이 경우 Overwrite을 여유 블록 0번까지 받는다. (모든 데이터, 여유 블록을 채운 경우) 이후 추가로 1개의 Overwrite 명령이 발생했을 때(최대한 데이터 지우기, 복사를 막기 위함) 여유 블록 0의 유효 데이터에 대한 LSN을 색인해서, 해당 데이터 블록에 대한 유효 데이터를 병합 블록(여유 블록 1)을 통해 복원해준다. 유효하지 않은 모든 데이터를 정리하면서 데이터 블록과 여유 블록 1의 논리 주소가 계속 변한다. 여유 블록 1번에 데이터가 쓰여질 때, 기존 데이터 블록이 비워지므로, 해당 데이터 블록을 여유 블록 1번으로 설정하고, 매핑 정보를 수정한다.

<동적 섹터 매핑 기법의 작동>



왼쪽은 동적 섹터 매핑에 앞서 설명한 정적 섹터 매핑과 같은 쓰기 요청을 받은 결과이며, 오른쪽의 경우 동적 섹터 매핑 기법에서 순차적으로 0~11까지의 데이터를 넣은 결과이다.

Case 1) 왼쪽 그림의 동적 할당에서는 매핑 테이블이 사전 결정되지 않고, 데이터가 들어올 때 빈 공간을 찾아서 할당시켜주는 기법이므로, 초기에 0~4까지의 순차적 쓰기 작업이 할당되었을 때, 마지막 PSN을 기록하는 변수를 선언하면서 0번 섹터부터 할당되었을 것이다. 이후 A', B'에 대한 요청이 들어왔을 때, 정적 할당과 다르게 마지막 PSN을 참조하여 비어있는 다음 페이지에 저장된다.((++last_PSN) %= numofsec)

0~11까지 사용자에게 보장해줘야 하는 공간이긴 하지만, 현재는 사용 중이지 않으며, 정적과 같이 하나의 블록에만 매핑 자유도를 제공하는 것은 동적 할당의 잠재력을 끌어내지 못한다고 생각한다. C', D', E, F에 대한 데이터 쓰기 요청을 동적으로 처리한다.

후속 쓰기 요청에 대해 여유 블록 0번까지 데이터를 받을 수 있으며, 이후 데이터 요청이 Flash Level에서 Overwrite인지 알 수 없으므로 유효하지 않은 데이터를 비워줘야 한다. 한 개의 블록은 존재해야 여러 블록의 데이터와 섞여서 들어오는 데이터 후속 요청을 받을 수 있으므로, 만약 10번에 대한 요청이나, Overwrite 요청이 들어올 때 유효하지 않은 데이터가 존재하는 블록 0번, 1번을 비우며 유효한 데이터는 여유 블록 1번에 저장한다.

Case 2) 오른쪽의 경우 last_PSN을 참조해서 빈 곳에 순차적으로 데이터가 들어가므로, 0~11까지의 데이터 쓰기 작업이 실시된다. 모든 섹터가 유효한 데이터이므로, Overwrite을 여유 블록 0번까지 받는다. 추가로 1개의 Overwrite 명령이 발생했을 때(최대한 데이터 지우기, 복사를 막기 위함) 여유 블록 0의 모든 유효 데이터에 대한 본래 위치를 계산해 병합 블록을 이용해(여유 블록 1번) 해당 데이터 블록에 존재하는 유효하지 않은 데이터를 모두 정리해준다. 이후 들어온 요청을 처리한다.

<정적/동적 섹터 매핑의 비교>

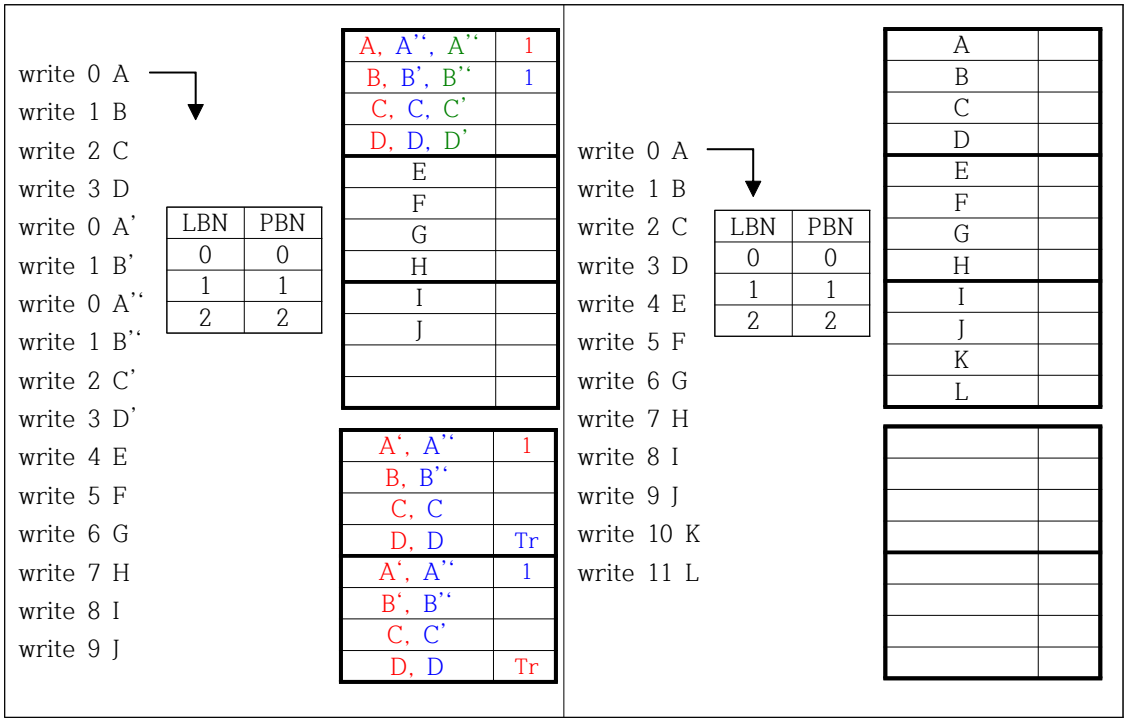
제안한 기법은 DRAM과 같은 캐시 공간을 사용하지 않으며, 결국 정적/동적 모두 매핑 테이블의 수정이 가능하도록 제안하였으므로, 어떤 기준으로 정적/동적 섹터 할당을 구분하였는지 설명하겠다.

정적의 경우, 먼저 사전 할당된 섹터 위치에 데이터를 쓰도록 하였으며, 동적의 경우 데이터 할당 시 비어있는 위치를 동적으로 찾아서 섹터 위치를 할당하도록 구상한 기법이다.

여기서 내 기법은 Overwrite이 발생했을 때 할당의 자유도에 대한 웨어 레벨링의 편차에서 차이가 발생하며, 그 결과 정적 섹터 매핑의 경우 복잡한 후처리 과정이 없다면 고르지 못한 NAND의 사용이 예상되며, 추가로 정적 섹터 매핑의 경우 동적 섹터 매핑보다 Overwrite 시 빈번한 지우기 연산이 발생될 것으로 예상된다. 동적 섹터 매핑의 경우 병합 공간을 제외한 모든 공간을 Write, Overwrite 공간으로 혼용해서 사용하므로, NAND의 사용량이 높을 경우에는 Write 시에도 저장 공간 확보를 위해서 Garbage Collection(복사, 지우기)연산이 발생될 것으로 예상되며, 이에 대한 처리시간이 정적 섹터 매핑보다 오래 걸릴 것으로 예상된다.

그리고 현재 내가 제안한 동적 섹터 기법은 결국 마지막으로 작성된 섹터를 기반으로 순차적으로 데이터를 쓰게 되는데, 모든 블록의 최소 1개의 섹터에서 Overwrite이 발생된다면, 블록이 고르게 지워지겠지만, 만약 해당 섹터에 유효한 데이터가 점유하고 있으면서, 해당 블록에 대한 Overwrite이 전혀 발생하지 않았다면, 영원히 지우지 않게 된다는 문제점이 있다. 결국 데이터가 어느 정도 들어있는 상태에서 특정 블록의 섹터를 제외한 집중적인 Overwrite가 발생된다면, 정적 섹터 매핑과 같이 NAND의 고른 사용이 불가능하다. 하지만 동적 할당 섹터 매핑 기법을 적용하였으며, 전반적인 블록 내 데이터가 적게 들어있으며, 여러 블록의 데이터가 Overwrite 될 경우, 유동적으로 사용할 수 있는 공간이 많으므로, 비교적 고르게 NAND를 사용하게 될 것으로 예상된다.
하지만 근본적으로 부실한 웨어 레벨링 기법임은 확실하다.

<정적 블록 매핑 기법의 작동>

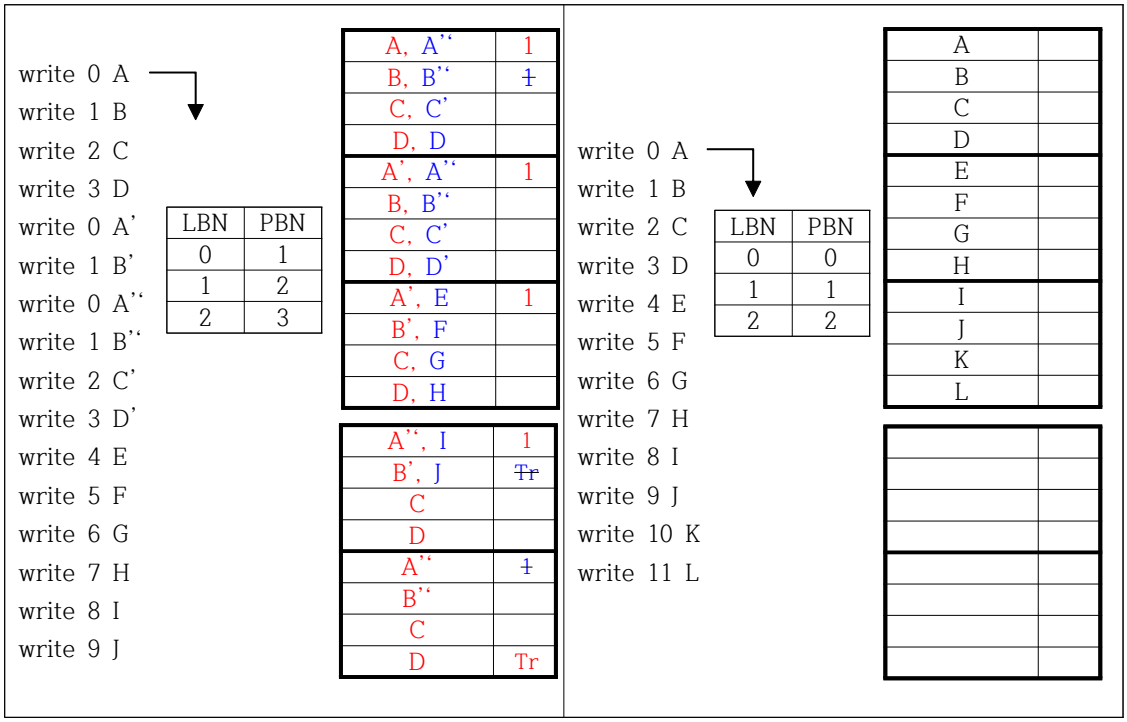


왼쪽의 경우, 앞에서 언급한 것과 같은 16개의 데이터 요청을 정적 블록 매핑에 적용한 결과이다. 오른쪽의 경우 12개의 순차적 데이터를 입력받았다.

Case 1) 정적 할당으로 인해, 논리 주소에 상응되는 물리 블록 주소를 사전 결정해둔 상태이다. 그러므로 0~3의 쓰기 요청에서, 물리 블록 0번에 모든 데이터가 쓰여진다. 이후 0번에 대한 추가 쓰기 요청이 발생했을 때, 데이터 쓰기 단위로 매핑되지 않으므로, 또 앞으로 어떤 데이터 요청이 들어올지 모르므로, 0번 섹터를 제외한 유효 데이터(현재는 모든 데이터)를 여유 블록 0번에 복사한다. 이후 매핑 정보를 여유 블록으로 수정해주면서 요청에 대한 데이터를 받게 된다. 그리고 Overwrite한 여유 블록 0번에 대한 변수를 선언해 데이터 블록 0번 주소를 저장해둔다. (중복 매핑을 피하기 위함) 이후 1번에 대한 추가 쓰기 요청이 들어왔을 때, 방금 작업과 같이 여유 블록 1번에 유효 데이터를 복사해주며, 이후 요청된 데이터를 저장하게 된다.(여유 블록 1번에 대한 기록 변수는 여유 블록 0번의 PBN이 된다.)
후속 덮어쓰기 요청에서 더는 쓸 수 있는 공간이 없으므로, 모든 블록의 유효하지 않은 데이터를 정리해준다. 데이터 블록 0번, 여유 블록 0번이 선택된다. 후속 요청에 대해 여유 블록 1번을 참조해서 유효한 데이터를 복사해오며, 요청된 데이터를 받는다. 동시에 매핑 정보도 수정해준다. 이후 4~9번에 대한 쓰기 요청에서는 해당 위치(offset)가 한 번도 쓰여진 적이 없으므로, 매핑 테이블을 참조해서 데이터 쓰기 요청을 마친다.

Case 2) 순차적으로 데이터 12개가 들어왔다면, 별도의 처리 없이 Overwrite를 여유 블록 1번까지 받는다. 이 과정에서 상응되는 기존의 PBN이 여유 블록 0, 1에 대한 기록 변수에 저장되어 있을 것이다. (매핑되지 않은 PBN)
이후 Overwrite이 추가로 발생되었다면, 모든 유효하지 않은 블록을 비우며 공간을 확보한다. 여유 블록 0번, 1번에 해당하는 변수의 PBN에 Overwrite 데이터, 유효 데이터를 받고, 매핑 정보를 수정해주며, 기록 변수를 수정해준다.

<동적 블록 매핑 기법의 작동>



왼쪽의 경우, 16개의 데이터 요청을 동적으로 처리한 결과이며, 빨강이 NAND의 첫 번째 쓰기 처리, 파랑색이 두 번째 쓰기 처리를 말한다.

Case 1) NAND가 비어있으므로, 0번 블록에 요청 0~3을 입력받아 처리한다. 동시에 마지막으로 사용한 물리 블록 주소를 기록한다.

이후 0, 1, 0, 1에 대한 요청을 여유 블록 1까지 입력받으며, 그리고 후속 쓰기 요청을 대비하여, 이 과정에서 유효하지 않는 블록 0, 1, 2, 여유 블록 0이 비어진다.

후속 쓰기 요청에 대해서 다음 블록을 구하는 공식인 (++last_PBN) %= numofblk을 통해 데이터 블록 0번부터 순차적으로 받게 되며, 이후 8번 섹터인 I까지 쓰여진 후, 섹터 9번에 대한 쓰기 요청을 마친다. 다음 요청이 Overwrite이라면, 유효하지 않은 블록을 비워줘야한다.

Case 2) 순차적으로 데이터 12개가 들어왔으며, 이후 Overwrite 요청이 들어왔을 때, 우선 여유블록 0번, 1번에 2번의 Overwrite를 받는다. 여기서 추가로 Overwrite이 더 발생해서 데이터 써야한다면, 유효하지 않은 모든 데이터 블록을 비워줘야 한다. 이후 추가로 발생된 Overwrite 요청을 처리하기 위해, 비어진 블록에 순차적으로 데이터를 받으며, 기존 데이터 블록의 유효한 데이터를 모두 복원해줌으로 쓰기 요청을 진행한다.

+) 블록 정적/동적 모두 할당 공간을 추가로 할당할 수 있지만, 이로 인해 결국 지우기 연산(Garbage Collection)이 자주 발생 될 것이며, 동적 블록 할당의 경우 설계된 웨어 레벨링이 제대로 작동하지 않을 것이다.

<정적/동적 블록 매핑의 비교>

블록 매핑의 특징으로 매번 Overwrite가 발생될 때 기존 데이터 블록의 유효한 데이터를 복사해줘야 하기 때문이다. (Logical Continuity 보장)

내가 제안하는 정적 블록 매핑 기법의 경우, 매핑 테이블의 수정이 가능하다는 가정으로 작성되었으며, 이를 기반으로 초기에는 사전 할당된 블록에 데이터를 작성하며, Overwrite가 발생했을 때, 블록 매핑을 기반으로 설계한 프로그램이므로, Logical Continuity를 보장하기 위해, 여유 블록에 기존 유효한 데이터를 새롭게 할당한 블록에 무조건 복사해주어야 한다.

이후 추가 요청된 데이터를 해당 오프셋에 가져옴으로 기본적인 데이터 덮어쓰기가 완료된다. 여기서 내가 제안한 정적과 동적의 성능 차이점은 섹터에서 언급한 것과 같은 매핑 자유도에 따른 고른 NAND 쓰기, 블록 단위 매핑 기법 특성으로 웨어 레벨링 부분에 대한 고려가 훨씬 잘 되어 있다는 것이다.

그 결과 정적 블록 할당의 경우, 데이터 블록과 여유 블록 2개를 돌아가면서 추가 요청된 데이터를 처리하므로, 집중적인 Overwrite이 발생되었을 때, 해당 블록 3개만 마모되게 된다.

그러나 내가 고안한 동적 블록 할당의 경우, Overwrite 데이터와 유효한 쓰기 요청에 대한 데이터를 순차적으로 모든 블록을 할당시키도록 해서 고른 NAND 마모 부분에서 매우 유리하다는 것을 알 수 있다.

해당 웨어 레벨링 기법은 동적 섹터와 같이 순차적으로 모든 블록을 마모시키도록 간구되었으므로, 결국 해당 블록에 대한 Overwrite가 더는 발생하지 않는다면, 기존의 유효한 데이터가 해당 블록을 선점하고 있을 것이므로, 모든 블록에 대한 균일한 지우기 연산이 발생되지 않는다.

물론 블록 매핑 기법으로 Overwrite를 모두 받는 것이 근본적으로 큰 Erase를 발생시키는 것이 문제지만, 정적의 경우 3개의 블록을 집중적으로 지우게 되며, 동적의 경우 순차적으로 최대한 모든 블록을 비우려고 하기 때문에 동적 블록 할당의 경우 정적 블록 할당보다 훨씬 장기간 사용할 수 있을 것이다. 그리고 선점하는 데이터로 인해 동적 섹터 할당 기법의 웨어 레벨링과 같은 문제점이 발생된다.