- 2025 전기 졸업과제 중간보고서 -

딥러닝을 이용한 SSD 성능 개선 연구

지도교수 : 안성용

분과명 : D

팀명 : SSDeep Learning

201924544 이준형

201914502 강인석

202155537 김지수

목차

1. 과제 목표

본 연구는 SSD 내부의 Flash Translation Layer(FTL)가 정적 정책 기반으로 동작하며 실제 워크로드 변화에 대응하지 못해 발생하는 쓰기 증폭과 지연 시간 증가, 수명 단축 문제를 해결하기 위해 기획되었다. FTL은 논리 주소와 물리 주소 간 매핑, 가비지 컬렉션(GC), 웨어 레벨링, 불량 블록 관리 등의 핵심 기능을 수행하지만, 정적 정책만으로는 반복적이거나 특징적인 I/O 패턴을 효율적으로 처리하기 어렵다.

이에 본 과제에서는 SNIA IOTTA 리포지터리[1]에서 제공하는 YCSB 블록 트레이스를 딥러닝 기법으로 학습·분석하여, SSD가 워크로드 특성에 능동적으로 대응할 수 있는 예측 모델을 개발하고자 한다. 구체적으로 다음과 같은 과제 수행 과정을 거쳤다.

- 데이터 전처리 및 패턴 분할
  – YCSB 블록 트레이스를 SimpleSSD-SA[2] 시뮬레이터 형식으로 변환
  – 대용량 I/O 요청을 SSD 페이지(8섹터) 단위로 분할하여 해시테이블에 누적
- Hotness 예측 모델 학습
  – 페이지별 접근 통계(평균 시간 간격, 표준편차, 요청 크기, 마지막 수명)로 4단계 Hotness(Cold→Hot) 레이블링
  – Stacked-LSTM을 이용해 과거 시퀀스 기반 다음 Hotness 예측 모델 구축
- 오프라인 레이블링 및 시뮬레이션 연동
  – 학습된 모델로 페이지별 쓰기 요청에 Hotness 레이블 부여
  – SimpleSSD-SA 시뮬레이션에 예측 레이블을 적용하여 GC 동작 및 성능 변화를 평가

최종적으로 본 연구는 예측된 I/O 패턴과 Hotness 정보를 FTL의 블록 할당 및 GC 우선순위 결정에 적용하여,

- 쓰기 증폭(Write Amplification) 감소
- 쓰기 지연 시간(Write Latency) 단축
- P/E 사이클 소모 절감을 통한 SSD 수명(Endurance) 연장

을 달성함으로써 SSD 성능 및 내구성을 향상시키는 것을 목표로 한다.

2. 갱신된 과제 추진 계획

업무	4				5				6				7				8				9			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
기획 및 설계																								
데이터 전처리																								
입출력 패턴 코드 분석																								
모델 개발																								
시뮬레이션 환경 구축 및 설정																								
FTL 개선 정책 구현																								
성능 측정																								
기존 FTL VS 딥러닝 기반 FTL 비교																								

3. 구성원별 진척도

3.1 구성원별 역할

이준형	데이터 전처리 입출력 패턴 코드 분석 모델 개발
강인석	시뮬레이션 환경 구축 및 설정 성능 측정(W/A) Hotness 레벨 트레이스용 FTL 정책 개선 성능 비교
김지수	시뮬레이션 환경 구축 및 설정 FTL 개선 정책 구현 성능 측정 기존 FTL VS 딥러닝 기반 FTL 비교

3.2 구성원별 진척도

이준형	데이터 전처리(완료) 입출력 패턴 코드 분석(완료) 모델 개발(진행중)
강인석	시뮬레이션 환경 구축 및 설정(완료) FTL 개선 정책 구현 성능 측정(W/A)(완료) 기존 FTL VS 딥러닝 기반 FTL 비교
김지수	시뮬레이션 환경 구축 및 설정 (완료) page mapping 코드 분석 (완료) 기존 FTL VS 딥러닝 기반 FTL 비교

4. 보고 시점까지의 과제 수행 내용 및 중간 결과

4.1 데이터 전처리 및 입출력 패턴 코드 분석

4.1.1 YCSB → SimpleSSD 트레이스 변환

YCSB에서 제공된 블록 단위 I/O 트레이스를 SimpleSSD-SA 시뮬레이터가 요구하는 형식으로 변환하는 과정은, 실험 재현성과 시뮬레이터 입력 간소화를 보장하기 위한 핵심 전처리 단계이다. 이 과정에서는 원본 트레이스의 불필요한 식별자와 부가 정보를 제거하고 통일된 형식으로 정제하며, 타임스탬프는 초 단위 기준으로 소수점 이하 9자리까지 패딩하여 시간 정밀도를 향상시키고, Operation Type은 RS와 WS 같은 중간 유형을 단순히 R과 W로 매핑하여 처리 효율을 높인다.

이를 보다 구체적으로 적용한 변환 단계에서는, 시뮬레이터의 일관된 동작을 위해 Device, Seq, PID 필드에 각각 "8,0", 0, 0의 더미값을 부여하여 외부 환경 정보에 대한 의존성을 제거한다. Timestamp 필드는 초 단위 값의 소수점 이하를 9자리까지 패딩 처리함으로써 I/O 요청 간 시간 간격을 정밀하게 반영하고, Trace Action은 D(issued)만 유지, Operation Type은 R/RS를 R로, W/WS를 W로 단순화하여 시뮬레이터의 입력 범위를 읽기와 쓰기 두 가지 유형으로 제한한다.

이러한 변환을 통해 Record ID, 원본 PID, Process Name과 같은 불필요한 메타데이터는 제거되고, 결과적으로 입력 데이터 크기는 줄어들며 파싱 및 처리 로직의 복잡도도 감소한다. 한편, 실제 요청의 물리 주소와 크기를 나타내는 Sector, '+', Size 필드는 원본 값을 유지함으로써, 시뮬레이터가 원래의 I/O 워크로드 특성을 정확하게 반영할 수 있도록 한다.

입출력 데이터 필드 변환 정리

YCSB 입력 필드	SimpleSSD-SA 출력 필드	변환 방식 및 설명
Device	Device	Major,Minor 번호를 “8,0”의 더미값으로 고정
CPU	CPU	값 변경 없이 그대로 사용
RecordID	Seq	값 제거, 출력을 “0”(더미값)으로 고정
Timestamp	Timestamp	초 단위 값에 대해 소수점 이하 9자리까지 패딩(예: 0.000005462 → 0.000005462)
PID	PID	값 제거, 출력을 “0”(더미값)으로 고정
Action	Action	D(issued)만 유지, 그대로 출력
Op	Op	R/RS → R, W/WS → W로 단순화
Sector	Sector	값 변경 없이 그대로 사용
+	+	값 변경 없이 그대로 사용
Size	Size	값 변경 없이 그대로 사용
ProcessName	- (없음)	출력에서 완전히 제거

입력 데이터 예시 (YCSB)

Device	CPU	RecordID	Timestamp	PID	Action	Op	Sector	+	Size	ProcessName
259,2	0	5	0.000005462	4020	D	RS	282624	+	8	[java]
259,2	0	11	0.013363110	4020	D	RS	286720	+	128	[java]

출력 데이터 예시 (SimpleSSD-SA)

Device	CPU	Seq	Timestamp	PID	Action	Op	Sector	+	Size
8,0	0	0	0.000005462	0	D	R	282624	+	8
8,0	0	0	0.013363110	0	D	R	286720	+	128

4.1.2 SimpleSSD 트레이스 → 페이지별 분할

SimpleSSD-SA 형식의 트레이스를 페이지 단위로 분할하는 단계는, 대용량 I/O 요청을 SSD의 물리적 페이지 경계에 맞춰 세분화하여 이후 페이지별 해시테이블 업데이트 및 Hotness 레이블링 절차에 바로 활용할 수 있는 형태로 변환하는 과정이다. 구체적으로, 요청 크기가 기본 페이지 단위인 8섹터의 배수(예: 256섹터, 즉 페이지 32개)로 주어지면 이를 8섹터씩 나누고, 각 서브요청의 시작 LBA는 원본 Sector 값에 '페이지 인덱스 × 8'을 더해 계산하며(Size는 8로 고정), 마지막 페이지는 남은 섹터 수만큼 크기를 조정한다. 이로써 하나의 대용량 요청은 동일한 Timestamp, Seq, Op를 유지한 채 여러 페이지별 요청으로 분해되고, 분할된 각 요청이 페이지 단위 처리 로직에 즉시 투입될 수 있도록 준비된다.

하나의 대용량 I/O 요청(256섹터)은 기본 페이지 크기인 8섹터 단위로 총 32개로 분할되며, 각 분할된 요청은 모두 동일한 Timestamp, Seq, Op 값을 유지한 채 개별 페이지 시작 LBA와 Size(8섹터)로 재정의되어 32개의 출력 레코드로 생성된다.

입력 데이터 예시 (SimpleSSD-SA 트레이스)

Device	CPU	Seq	Timestamp	PID	Action	Op	Sector	+	Size
8,0	0	0	150.122992736	0	D	R	381785824	+	256

출력 데이터 예시 (페이지별 분할 트레이스)

Device	CPU	Seq	Timestamp	PID	Action	Op	Sector	+	Size
8,0	0	0	150.122992736	0	D	R	381785824	+	8
8,0	0	0	150.122992736	0	D	R	381785832	+	8
8,0	0	0	150.122992736	0	D	R	381785840	+	8
8,0	0	0	150.122992736	0	D	R	381785848	+	8
8,0	0	0	150.122992736	0	D	R	381785856	+	8
8,0	0	0	150.122992736	0	D	R	381785864	+	8
8,0	0	0	150.122992736	0	D	R	381785872	+	8
8,0	0	0	150.122992736	0	D	R	381785880	+	8
8,0	0	0	150.122992736	0	D	R	381785888	+	8
8,0	0	0	150.122992736	0	D	R	381785896	+	8

8,0	0	0	150.122992736	0	D	R	381785904	+	8
8,0	0	0	150.122992736	0	D	R	381785912	+	8
8,0	0	0	150.122992736	0	D	R	381785920	+	8
8,0	0	0	150.122992736	0	D	R	381785928	+	8
8,0	0	0	150.122992736	0	D	R	381785936	+	8
8,0	0	0	150.122992736	0	D	R	381785944	+	8
8,0	0	0	150.122992736	0	D	R	381785952	+	8
8,0	0	0	150.122992736	0	D	R	381785960	+	8
8,0	0	0	150.122992736	0	D	R	381785968	+	8
8,0	0	0	150.122992736	0	D	R	381785976	+	8
8,0	0	0	150.122992736	0	D	R	381785984	+	8
8,0	0	0	150.122992736	0	D	R	381785992	+	8
8,0	0	0	150.122992736	0	D	R	381786000	+	8
8,0	0	0	150.122992736	0	D	R	381786008	+	8
8,0	0	0	150.122992736	0	D	R	381786016	+	8
8,0	0	0	150.122992736	0	D	R	381786024	+	8
8,0	0	0	150.122992736	0	D	R	381786032	+	8
8,0	0	0	150.122992736	0	D	R	381786040	+	8
8,0	0	0	150.122992736	0	D	R	381786048	+	8
8,0	0	0	150.122992736	0	D	R	381786056	+	8
8,0	0	0	150.122992736	0	D	R	381786064	+	8
8,0	0	0	150.122992736	0	D	R	381786072	+	8

4.1.3 LSTM 학습 데이터 생성

LSTM 학습 데이터는 페이지 단위 분할 트레이스를 순차적으로 처리하며 네 개의 단계로 생성된다. 각 단계의 목적과 내부 동작은 다음과 같다.

페이지 해시 테이블 생성

트레이스 파일의 각 레코드를 읽으면서 LPN을 키로 하는 해시 테이블을 구축·갱신한다. 해시 테이블의 값(value)은 아래 여섯 개 필드를 갖는 구조체이며, 새 LPN이 등장할 때마다 초기화하고 이후 재등장 시마다 갱신한다.

- access_count: 해당 페이지에 대한 총 쓰기 접근 횟수
- time_gaps: 이전 접근 시점과 현재 접근 시점 간 시간 간격 리스트
- request_sizes: 각 접근 시 요청된 페이지 크기 리스트
- last_access_time: 가장 최근 쓰기 요청 시점
- last_lifetime: 바로 이전 접근 시점까지의 수명(시간 차이)
- access_history: 각 접근 시점별 위 정보의 스냅샷을 순서대로 저장

초기화 시 접근 횟수를 1로, time_gaps는 빈 리스트로, request_sizes는 현재 크기 한 건으로, last_access_time을 현재 시각으로, last_lifetime을 0으로 설정한다. 동일 LPN 재등장 시 access_count를 1 증가시키고, gap = current_timestamp − last_access_time을 time_gaps에 추가한 뒤 last_lifetime에 gap을 기록, request_sizes에 현재 요청 크기를 추가하고 last_access_time을 갱신하며, 갱신된 세 가지 필드(time_gaps, request_sizes, last_lifetime)를 access_history에 스냅샷 형태로 보관한다.

네 가지 통계 특성(feature) 추출

해시 테이블 작성 완료 후, access_count가 2 이하인 페이지(노이즈 페이지)는 통계 정보가 부족하므로 미리 제외한다. 남은 각 페이지에 대해 다음 네 가지 통계치를 계산해 4차원 피처 벡터를 구성한다.

특성명	계산 방법	설명
시간 간격 평균	mean(time_gaps)	페이지 접근 주기의 평균
시간 간격 표준편차	std(time_gaps)	접근 주기 변동성
요청 크기 평균	mean(request_sizes)	페이지별 요청 크기 평균
마지막 수명	last_lifetime	직전 접근 간 시간 차이

K-means 군집화 (K=4)

추출된 피처 벡터를 Pandas DataFrame으로 변환한 뒤 StandardScaler로 정규화하고, 관성(inertia) 곡선 분석을 통해 K=4를 결정하여 K-means 클러스터링을 수행한다. 이때 각 페이지에 0(Cold)부터 3(Hot)까지의 Hotness 레이블이 자동 할당되며, 네 개의 클러스터 중심값은 별도 파일로 저장된다.

LSTM 입력 시퀀스 생성

과거 세 번 연속된 특성 벡터를 길이 3의 슬라이딩 윈도우 형태로 묶어 (샘플 수, 3, 4) 크기의 3차원 배열을 생성한다. 각 시퀀스에 대응하는 정답 레이블은 그 다음 시점의 Hotness 값으로 구성되며, 이렇게 준비된 데이터를 이용해 Stacked-LSTM 모델을 학습함으로써 과거 페이지 접근 패턴으로부터 미래의 Hotness를 예측할 수 있다.

4.1.4 오프라인 레이블링

학습된 Stacked-LSTM 모델을 이용해 페이지별 분할 트레이스를 로드한 후, 읽기(Op=R) 요청과 쓰기(Op=W) 요청을 구분한다. 쓰기 요청에 대해서는 표준화된 LSTM 입력 시퀀스를 모델에 입력하여 0부터 3까지의 Hotness 클래스를 예측하고, 예측값을 Hotness_Label 필드에 추가한다. 읽기 요청은 별도의 예측 없이 Hotness_Label을 –1로 지정한다. 이후 원본 10개 필드 뒤에 Hotness_Label을 덧붙여 저장하며, 처리된 모든 레코드에 대해 읽기 요청은 –1, 쓰기 요청은 모델이 예측한 0~3 레이블을 갖는다. 마지막으로 전체 트레이스를 집계하여 읽기 요청 수와 Hotness 레벨별(0: 콜드, 1: 미디움, 2: 웜, 3: 핫) 레코드 수를 통계 리포트 형태로 출력한다. 이 오프라인 레이블링 과정을 통해 SimpleSSD-SA 시뮬레이션에 실제 I/O 워크로드에 부합하는 Hotness 정보를 간편하게 삽입할 수 있다.

입력 데이터 예시 (페이지별 분할 트레이스)

Device	CPU	Seq	Timestamp	PID	Action	Op	Sector	+	Size
8,0	0	0	150.122992736	0	D	W	381785824	+	8
8,0	0	0	150.122992736	0	D	R	381785840	+	8

출력 데이터 예시 (레이블링된 트레이스)

Device	CPU	Seq	Timestamp	PID	Action	Op	Sector	+	Size	Hotness_Label
8,0	0	0	150.122992736	0	D	W	381785824	+	8	2
8,0	0	0	150.122992736	0	D	R	381785840	+	8	-1

4.2 SimpleSSD-SA 구조 분석 및 W/A 측정 기능 구현

SimpleSSD-SA는 SSD의 내부 동작, 특히 FTL(Flash Translation Layer) 및 NAND 계층의 동작을 시뮬레이션하기 위해 설계된 오픈소스 프레임워크이다. 기본적으로 다양한 SSD 워크로드를 처리할 수 있는 구조를 갖추고 있지만, 쓰기 증폭(Write Amplification, W/A)과 같은 핵심 성능 지표를 직접적으로 제공하지는 않는다. 본 보고서에서는 SimpleSSD-SA에 W/A 측정 기능을 새롭게 구현하여, 논리/물리 쓰기량을 기반으로 시뮬레이션 정확도를 향상시킬 예정이다.

4.2.1 SimpleSSD-SA 구조 및 모듈 개요

SimpleSSD-SA는 모듈화된 설계를 통해 각 구성 요소의 독립성을 보장하고, 시뮬레이션의 유연성과 확장성을 높였다.

구조 요약 표

구성 요소 (클래스/모듈)	**핵심 역할**
ConfigReader	시뮬레이션 설정 파일 파싱 및 로드
Engine	시뮬레이션 시간 진행 관리 및 이벤트 스케줄링
TraceReplayer	외부 트레이스 파일 읽기, I/O 요청 생성 및 논리 쓰기량 집계
BIL::Driver	시뮬레이터의 상위 계층(호스트)과 하위 계층(FTL/SSD) 간 인터페이스 제공
FTL (예: PageMapping)	논리 주소를 물리 주소로 매핑, 가비지 컬렉션 및 물리 쓰기량 집계

전체 동작 흐름 (Pseudocode)

SimpleSSD-SA의 main 함수는 시뮬레이션 환경을 초기화하고, I/O 요청을 처리하며, 시뮬레이션 종료 후 통계를 출력하는 일련의 과정을 담당한다.

```
procedure main():
    // 1. 시뮬레이션 설정 읽기 및 초기화
    configReader = 새 ConfigReader(설정 파일 경로)
    SimpleSSD 엔진 초기화(configReader로부터 설정 로드)
    driver = 새 BIL::Driver(엔진 인스턴스, 설정 객체)

    // 2. I/O 요청 처리를 위한 객체 생성
    blockIOEntry = 새 BlockIOEntry()
      ioGenerator = 새 TraceReplayer(엔진, blockIOEntry, 종료 콜백 함수,
configReader)

    // 3. FTL 통계 연결 (FTL이 사용되는 경우)
    if (driver가 FTL 인터페이스를 포함):
        ioGenerator의 FTL 통계 객체 설정(driver의 내부 FTL 객체)
    end if

    // 4. 시뮬레이션 이벤트 루프 실행
    while (엔진에 처리할 다음 이벤트가 존재):
        엔진의 다음 이벤트를 처리하고 시간 진행
    end while

    // 5. 시뮬레이션 종료 후 통계 출력 및 자원 정리
    시뮬레이션 종료 후:
        ioGenerator의 통계 출력
        엔진의 통계 출력
        각종 할당된 자원 정리 (드라이버, I/O 생성기 등)
end procedure
```

4.2.2 주요 클래스 설명

`TraceReplayer` 클래스

역할: 외부 트레이스 파일(예: YCSB 트레이스)을 읽고 파싱하여 시뮬레이터에 제출 가능한 I/O 요청(BlockIO)으로 변환한다. 사용자로부터 발생한 논리적 쓰기량(user write bytes)을 집계하는 역할을 담당한다.

주요 자료 구조 : `TraceReplayer` 내부에는 `uint64_t` 타입의 `write_bytes` 변수가 선언되어 사용자가 요청한 총 논리 쓰기량(바이트)을 누적한다. 또한, 트레이스 파일을 파싱할 때 각 I/O 요청의 정보를 임시 저장하는 `TraceLine` 구조체(`tick`, `offset`, `length`, `type`, `hotness_level` 등의 필드 포함)를 활용한다.

주요 함수:`procedure TraceReplayer::submitIO()`: 파싱된 트레이스 데이터를 기반으로 I/O 요청을 생성하고 시뮬레이터에 제출하는 핵심 함수이다. 이 함수 내부에서 쓰기 요청의 논리적 바이트 수를 **`write_bytes`**에 추적한다

```
procedure TraceReplayer::submitIO():
    // ... (I/O 요청 객체인 'bio'에 linedata의 정보를 설정) ...

    if (linedata.type == BIL::BIO_WRITE): // 쓰기 요청인 경우
        write_bytes = write_bytes + linedata.length // 논리 쓰기량 누적
    end if

     bioEntry.submitIO(bio) // I/O 요청을 BlockIOEntry를 통해 시뮬레이터에
제출
    // ...
end procedure
```

PageMapping 클래스 (FTL 계층의 일부)

역할: FTL(Flash Translation Layer) 계층의 핵심 구성 요소로, 논리 주소(LPN)를 물리 주소(PPN)로 매핑하고, 가비지 컬렉션(GC) 및 마모도 평준화(Wear Leveling) 등 실제 SSD의 핵심 동작을 시뮬레이션한다. 이 과정에서 발생하는 물리적 쓰기량(device_write_bytes)을 추적한다.

주요 함수 : PageMapping::writeInternal(LPN, Size): 특정 논리 페이지 번호(LPN)에 대한 쓰기 요청을 처리하며, 이 과정에서 실제 플래시 메모리에 기록되는 물리적 바이트 수를 계산하고 누적할 수 있다.

```
procedure PageMapping::writeInternal(LPN, Size):
    // ... (논리 페이지를 물리 페이지로 매핑하는 과정) ...

    if (새로운 물리 페이지에 데이터가 기록되면):
            device_write_bytes = device_write_bytes + Size // 물리 쓰기량
누적
    end if

    // ... (매핑 테이블 갱신, 가비지 컬렉션 트리거 등) ...
end procedure
```

Engine 클래스

역할: 시뮬레이션의 전체적인 시간 진행을 관리하고, 등록된 이벤트들을 스케줄링하며 실행한다. I/O 요청 제출, I/O 완료, 통계 출력 등 모든 시뮬레이션 이벤트는 Engine에 의해 관리된다.

4.2.3 W/A 측정 및 출력 기능

쓰기 증폭(Write Amplification, W/A)은 SSD의 성능과 수명에 직접적인 영향을 미치는 중요한 지표이다. SimpleSSD-SA는 논리 쓰기량과 물리 쓰기량을 정확하게 추적하여 W/A를 계산하고 출력하는 기능을 구현했다.

논리/물리 쓰기량 추적 방법

1. 논리 쓰기량 추적: TraceReplayer 클래스의 submitIO() 함수에서 트레이스 파일로부터 읽어 들인 각 쓰기 요청의 바이트 길이(linedata.length)를 TraceReplayer 내부의 uint64_t write_bytes 변수에 누적한다. 이는 사용자가 SSD에 요청한 순수한 쓰기 데이터 양을 나타낸다.
2. 물리 쓰기량 추적: FTL 계층의 PageMapping 클래스 내부에서 데이터를 실제 NAND 플래시 메모리에 기록할 때마다, 즉 PageMapping::writeInternal()과 같은 함수 호출 시마다 실제 기록된 바이트 수를 uint64_t device_write_bytes 변수에 누적한다. 이 과정에는 가비지 컬렉션(GC) 등으로 인해 발생하는 내부적인 쓰기 오버헤드까지 모두 포함된다.

전체 W/A 측정 흐름 요약

1. Trace 파일 파싱 및 논리 쓰기량 집계: TraceReplayer가 트레이스 파일을 읽어 각 I/O 요청의 정보를 파싱하고, 쓰기 요청 시 논리 쓰기량(write_bytes)을 누적한다.
2. I/O 요청 FTL 전달: 파싱된 I/O 요청은 BIL::BlockIOEntry를 통해 FTL 계층(PageMapping)으로 전달된다.
3. FTL 처리 및 물리 쓰기량 집계: PageMapping 클래스는 전달된 쓰기 요청을 처리하며, 실제 NAND 플래시에 데이터가 기록될 때마다 물리 쓰기량(device_write_bytes)을 누적한다.
4. W/A 계산 및 출력: 시뮬레이션 종료 시 TraceReplayer는 누적된 write_bytes와 device_write_bytes 값을 가져와 W/A를 계산하고, printStats() 함수를 통해 최종 통계를 출력한다.

4.2.4 핫/콜드 구분 기반 GC 제어 구조 설계 예정

기존 SimpleSSD는 FTL의 GC(가비지 컬렉션) 정책이 고정되어 있어, 특정 페이지의 쓰기 패턴(Hot/Cold)에 따른 차등 정책 적용이 불가능하다. 본 과제에서는 트레이스 파일에 포함된 Hotness 레벨 정보를 FTL에 주입하여, GC 정책이 이를 기반으로 victim 블록을 보다 효율적으로 선택하도록 개선할 계획이다.

구체적으로는, PageMapping 계층에 updateHotnessMap(std::vector<int>) 인터페이스를 추가하고, TraceReplayer가 트레이스를 파싱하면서 각 LPN의 Hotness 정보를 해당 인터페이스를 통해 FTL에 전달하도록 구현할 예정이다. 이후 GC 시에는 이 Hotness 맵을 참조하여, Cold 페이지가 많은 블록을 우선 회수하는 방식으로 Wear-Leveling과 Write Amplification 저감을 동시에 도모한다.

이는 실시간 예측 기반은 아니지만, 예측된 결과를 정적 구조로 시뮬레이션 내에 반영함으로써 향후 딥러닝 모델 성능 평가 및 GC 정책 최적화에 활용할 수 있는 기반을 제공한다

5. 요구 조건 및 제약 사항 분석에 대한 수정 사항

5.1 입력 피처 및 레이블링 방법론 변경

5.1.1 입력 피처 변경 사항

모델 입력은 단순한 원시 트레이스 필드 대신, 각 페이지별 접근 패턴을 정량적으로 요약하는 네 가지 통계적 특성(시간 간격 평균, 시간 간격 표준편차, 요청 크기 평균, 마지막 수명)으로 구성하였다. 이러한 특성 추출 방식은 데이터 내 노이즈를 효과적으로 줄이고, Hotness 예측의 정확도를 높이는 데 기여한다.

5.1.2 레이블링 방식 변경 사항

Hotness 레이블링은 단순 빈도 기반 이진 분류에서 벗어나, 각 LPN의 다음 접근 간격(next_lifetime)을 활용한 K-means 클러스터링(K=4)으로 0(콜드)부터 3(핫)까지 네 단계로 세분화하였다. 이 다단계 분류 방식을 통해 블록 할당 및 GC 우선순위 결정 시 더욱 정밀한 제어가 가능해져, SSD 성능 개선 효과를 극대화할 수 있다.

5.2 시뮬레이터 활용에 따른 제한사항

SimpleSSD-SA는 Page-Level Mapping FTL 구조를 기반으로 설계되어 있어, 블록 단위 매핑, Hybrid Mapping, 또는 최신 NAND 특성(e.g., SLC/MLC/TLC 구분)에는 제한적이다. 또한 시뮬레이터 내 IO 처리 루프는 단일 쓰레드로 구성되어 병렬 쓰기와 병렬 GC 시나리오에 대한 정밀한 재현에는 제약이 존재한다. 이에 따라 본 연구의 딥러닝 기반 예측 결과를 반영한 GC 우선순위 조정 설계는 Page-Level Mapping에 특화된 방식으로 제한되며, 향후 다양한 FTL 구조에 대한 적용 가능성은 별도 고려가 필요하다.

6. 멘토의견서에 대한 대응 방안

6.1 용어정리

“보고서에서 사용되는 용어들을 추후 보고서 작성 시 이해할 수 있도록 정리하거나 설명해 준다면 좋은 보고서가 될 수 있을 것”이라는 피드백을 반영하여, 본 보고서에서는 주요 용어를 정리하고 설명하였다. 이를 통해 보고서의 이해도를 높이고, 향후 관련 내용을 작성하거나 참고할 때 도움이 되도록 하였다.

PPN	SSD 내부의 실제 플래시 메모리 셀에서 데이터가 저장되는 위치를 나타내는 고유 번호
LPN	논리 페이지 번호
Warm-up	SSD가 처음부터 빈 상태로 시작하지 않도록, 일정량의 데이터를 미리 쓰는 과정
Dummy write	시뮬레이션이나 실험을 위한 임의의 쓰기 동작을 수행
쓰기 증폭	유효 데이터보다 더 많은 양의 데이터가 쓰이는 현상

6.2 page mapping

“현재 사용하고자 계획 중인 시뮬레이터인 SimpleSSD의 FTL을 연구하거나, SimpleSSD에서 언급하고 있는 Page-level 매핑에 대한 학습을 진행하는 것이 좋다”는 피드백을 반영하여, 이론적 배경을 학습하고 분석하고자 한다.

SimpleSSD는 기본 FTL 방식으로 Page-level Mapping FTL 방식을 사용한다. page mapping 파일에는 생성자와 주요 입출력 함수인 write, read, trim 함수가 정의되어 있으며, 이들은 각각 내부적으로 writeInternal, readInternal, trimInternal과 eraseInternal 등의 함수들을 호출한다.

각 함수는 각각 쓰기, 읽기, trim 명령을 처리하며 내부적으로 Internal 함수들을 호출한다. 보고서에서는 목표와 관련된 주요 기능인 생성자, writeInternal, eraseInternal, doGarbageCollection 함수에 대해 중점적으로 설명하고자 한다.

생성자는 객체 생성 시 내부 상태를 초기화하며, 전체 물리 블록 수에 기반해 freeblocks를 구성하고, 전체 논리 페이지 수를 계산하여 매핑 테이블 공간을 미리 할당한다.

6.2.1 initialize

이 함수는 시뮬레이션 시작 전, 메모리 warm-up과 invalid page 생성을 수행하는 역할을 한다.

기초 변수 초기화

시뮬레이션 설정 파일에 따라 총 논리 페이지 수와 사전 할당할 비율을 계산한다. 이를 통해 시뮬레이션에 필요한 메모리 및 매핑 구조의 크기, 초기 상태를 결정한다. FillRatio는 시뮬레이션 시작 전에 SSD의 논리 페이지 중 얼마나 채워둘 것인지 결정하는 비율이고, InvalidPageRation는 이미 채운 논리 페이지들 중에서 얼마나 다시 덮어써서 invalid 페이지로 만들 것인지 결정하는 비율이다.

GC 여유 공간 확보 검사

너무 많은 페이지를 쓰고/무효화하면 GC를 바로 유발할 수 있으므로, GC 임계값을 고려하여 무효화 페이지 수를 조정하게 된다.

LPN 범위에 따라 sequential/random하게 dummy write를 수행하게 된다. 이 작업으로 이미 쓰여진 상태의 SSD를 시뮬레이션한다.

FILLING_MODE_0	논리 페이지를 0부터 순차적으로 채움	순차적으로 다시 overwrite
FILLING_MODE_1	쓰기 순차적으로 수행.	무효화는 랜덤 선택하여 overwrite
FILLING_MODE_2	쓰기 자체도 논리 페이지를 완전히 랜덤하게 선택함	무효화도 전체 논리 페이지 범위에서 랜덤하게 선택

Invalidation (무효화) 단계

GC가 유효하게 작동하려면 invalid 페이지가 존재해야 하므로 일부 LPN을 overwrite하여 invalid page 생성하게 된다. mode에 따라 순차 또는 랜덤하게 invalidation 수행한다.

```
procedure PageMapping::initialize()
    if (mode가 FILLING_MODE_0):
        for (i를 0부터 nPagesToInvalidate - 1까지 증가):
            tick 값을 0으로 초기화
            req.lpn에 i 값을 저장
            writeInternal 함수 호출 (req, tick, false) // 쓰기
         end for
    end if
end procedure
```

6.2.2 writeInternal

기존 매핑 무효화 (invalidate)

해당 LPN이 이미 존재한다면 기존의 물리 페이지 정보를 찾아 무효화한다. 이는 Overwrite 특성을 반영한 것이다. LPN이 처음 등장한 경우에는 새로운 매핑 테이블을 생성한다.

```
procedure PageMapping::writeInternal(req, tick, sendToPAL)
    mappingList ← 매핑 테이블에서 req.lpn에 해당하는 항목 검색
    // 타이밍 기록 초기화
    beginAt ← 0
    finishedAt ← tick
    readBeforeWrite ← false

    if (mappingList가 존재한다면):
        for (idx from 0 to bitsetSize - 1):
            조건1 ← 요청의 ioFlag[idx]가 true
            조건2 ← 랜덤 트윅이 비활성화됨

            if (조건1 or 조건2):
                mapping ← mappingList의 idx번째 매핑 정보

                if (mapping이 유효한 물리 위치라면):
                    block ← 해당 물리 블록 검색
                    block에서 invalidate(mapping.second, idx) 호출
                 // 기존 물리 페이지 무효화 (invalidate)
```

```
                end if
            end if
        end for
    end if
end procedure
```

쓰기 수행 및 매핑 테이블 갱신

실제 PPN을 결정하고 매핑 테이블을 갱신한다. 물리 블록 내 사용 가능한 페이지를 찾아 쓰기를 수행하게 된다.

```
procedure PageMapping::writeInternal(req, tick, sendToPAL)
    // . . .
    for (idx from 0 to bitsetSize - 1):
        if (요청의 ioFlag[idx]가 true이거나 랜덤 트윅이 비활성화된 경우):
            pageIndex ← 해당 블록에서 idx에 대한 다음 쓰기 위치 가져오기
            mapping ← mappingList의 idx 번째 항목
            beginAt ← tick

            write(pageIndex, req.lpn, idx, beginAt) 호출  // 쓰기 수행
        end if
    end for
end procedure
```

GC 수행 조건 확인 및 트리거

남은 Free Block의 비율이 임계값보다 작으면 GC를 수행하고, 유효 페이지 복사, 블록 erase 등이 이때 발생하게 된다.

```
procedure PageMapping::writeInternal(req, tick, sendToPAL)
    // . . .
    if (freeBlockRatio() < gcThreshold):
        if (sendToPAL == false):
            오류 발생
        end if

        리스트 ← 비어있는 블록 리스트 생성
        beginAt ← tick
        selectVictimBlock(리스트, beginAt) 호출
        // . . .
        doGarbageCollection(리스트, beginAt) 호출
    end if
end procedure
```

6.2.3 eraseInternal

블록 유효성 검사 후 내부적으로 erase()를 호출하여 상태 초기화하고 PAL 계층으로 명령을 보내 실제로 NAND에서 지우도록 시뮬레이션한다.

Wear-Leveling 고려한 재등록

freeBlocks는 erase count 오름차순으로 정렬된다. 블록의 erase 횟수가 threshold보다 작으면 다시 사용 가능하고, freeblcoks에 정렬된 위치로 다시 삽입되게 된다. (erase count 순으로) 이렇게 되면 erase count가 낮은 블록이 앞쪽에 유지되도록 정렬을 보장한다.

```
procedure PageMapping::eraseInternal(req, tick)
    // . . .
    if (erasedCount < threshold):
        iter ← freeBlocks의 끝 위치에서 시작

        // 역순 탐색
        while true do
            iter ← iter - 1

            if (iter의 eraseCount ≤ erasedCount):
                iter ← iter + 1  // 다시 앞으로 이동 (원래 위치로)
                break
            end if

            if (iter가 freeBlocks의 시작 위치라면):
                break
            end if
        end while
    end if

    // 찾은 위치에 블록 삽입 (eraseCount 기준 정렬 유지)
    freeBlocks에 iter 위치로 block을 이동 삽입
    nFreeBlocks ← nFreeBlocks + 1
end procedure
```

6.2.4 doGarbageCollection

가비지 컬렉션(GC)을 실제로 수행하는 핵심 함수로, 이 함수는 유효 페이지를 새 블록으로 복사하고, 원래 블록을 erase한 뒤, 다시 freeBlocks 리스트에 넣어 재사용 가능하게 만드는 과정을 담당한다. 읽기를 준비한 뒤 복사하고, 매핑 테이블을 업데이트 한 뒤에 원래 블록을 erase 하고 freeblocks로 반환한 뒤에 tick을 업데이트 하게 된다.

```
procedure PageMapping::doGarbageCollection(blocksToReclaim, tick)

    // 유효 페이지 읽기 요청 생성
    req.blockIndex ← block의 블록 번호
    req.pageIndex ← pageIndex
    req.ioFlag ← bit
    readRequests에 req 추가

    //  매핑 테이블 업데이트
    newBlockIdx ← freeBlock의 블록 번호

    for (idx ← 0 to bitsetSize - 1):
        if (bit[idx]가 true라면):
            // 기존 페이지 무효화
            block.invalidate(pageIndex, idx)

            // 매핑 테이블 항목 가져오기
            mappingList ← table에서 lpns[idx]에 해당하는 항목 검색

            if (mappingList가 없다면):
                오류 발생
            end if

            // DRAM에서 읽기 요청
            pDRAM.read(mappingList 포인터, 8 × ioUnitInPage, tick)
        end if
    end for
end procedure
```

6.3 FTL 동작과 SSD의 성능 및 수명의 상관관계

“FTL의 정적인 동작을 분석하고 데이터 패턴을 분석하여 그것이 어떤식으로 쓰기 증폭을 감소시키고 SSD의 성능 및 수명을 증가시키는지 그 상관관계에 관해서도 설명이 있으면 좋겠다”는 피드백에 따라 부가적인 설명을 덧붙이고자 한다. SimpleSSD의 기본 FTL은 Page-level Mapping 방식을 사용하며, 이는 각 LPN을 개별 PPN으로 매핑하는 방식이다. 이 구조는 높은 유연성과 낮은 읽기 지연을 제공하지만, 덮어쓰기 시마다 새로운 페이지를 할당하고 기존 페이지를 무효화해야 하므로, 쓰기 증폭이 유발되기 쉽다.
정적인 구조 분석 결과, FTL은 쓰기 요청 시 다음과 같은 고정된 절차를 따른다.

1. 매핑 테이블 탐색 및 기존 페이지 무효화
   이미 존재하는 LPN에 대해선 이전 PPN을 무효화 처리한다.
2. 새로운 페이지 할당 및 매핑 테이블 갱신
   유휴 페이지를 할당한 후, 해당 정보를 매핑 테이블에 갱신한다.
3. 임계값 이하일 경우 GC 수행
   Freeblock 비율이 낮아지면 가비지 컬렉션을 트리거하여 유효 페이지를 새로운 블록으로 복사하고, 원래 블록을 지운 후 재사용 대기 상태로 이동시킨다.

이처럼 정해진 방식의 쓰기 처리와 GC 수행 구조는 워크로드의 특성(예: 반복적인 덮어쓰기, Hot 데이터 집중 등)에 따라 성능 및 수명에 큰 영향을 미칠 수 있다.

이에 본 연구에서는 이러한 정적인 흐름을 기반으로, LPN의 데이터 특성(Hot/Warm/Cold)에 따라 물리 블록을 그룹화하여 분리 저장하는 방식으로 최적화 기법을 제안하게 된 것이다. 이러한 방식을 사용하면 아래와 같은 효과를 얻을 수 있다.

- GC 효율성 증가
  Hot 데이터를 별도의 블록에 모아 저장하면, GC 시 유효 페이지 복사 대상이 주로 Hot 데이터 블록에 한정된다. 따라서 Cold 데이터 블록은 거의 변경되지 않아 GC 대상이 적어지고, 불필요한 데이터 이동이 줄어든다. 이로 인해 GC에 소요되는 시간과 오버헤드가 감소하여 전체 I/O 지연이 줄어든다.

- 쓰기 증폭 감소

  데이터 특성에 맞게 블록을 분리하면, 덮어쓰기가 집중되는 Hot 데이터 블록에서만 무효 페이지가 많이 발생하고, Cold 데이터 블록은 안정적으로 유지된다. 따라서 전체적으로 불필요한 쓰기와 블록 지우기가 줄어들어 쓰기 증폭이 감소한다.

- Wear-leveling 및 수명 향상

  Hot 데이터가 집중된 블록만 자주 지워지고, Cold 데이터 블록은 상대적으로 덜 지워지므로, 블록별 마모가 균등하게 분산된다. 이는 SSD 수명을 연장하는 효과를 가져온다.

- 매핑 테이블 및 내부 처리 효율 증가

  데이터 특성에 따른 분리 저장은 매핑 테이블 갱신 및 내부 관리가 보다 예측 가능하고 효율적으로 이루어지게 하여, FTL 처리 성능 향상에도 기여하게 된다.

7. 참고문헌

[1] "IOTTA Repository: Input/Output Traces, Tools, and Analysis," SNIA. [Online].
Available: https://iotta.snia.org/

[2] "SimpleSSD: Open-Source Licensed Full-System SSD Simulator," SimpleSSD,
https://docs.simplessd.org/en/v2.0.12/

[3] AGRAWAL, Nitin, et al. Design tradeoffs for {SSD} performance.
In: 2008 USENIX Annual Technical Conference (USENIX ATC 08). 2008

[4] 윤성준. "머신러닝 기반 Hot/Cold 페이지 분류를 통한 SSD 쓰기 증폭 개선 연구." 국내석사학위논문 부산대학교 대학원, 2024. 부산