되돌리기-L2P 테이블을 이용한 FTL-수준 트랜잭션 지원

이두기, 노홍찬, 박상현 *
연세대학교 컴퓨터과학과
e-mail : {edoogie, fallsmal, sanghyun}@cs.yonsei.ac.kr

An FTL-level Transaction Support using Undo-L2P Table

Doogie Lee, Hongchan Roh, Sanghyun Park*
Dept. of Computer Science, Yonsei University

요 약

NAND 플래시 메모리 기반 저장장치들은 기존의 일반적인 저장장치들과는 다른 독특한 저장방식을 가지고 있어서 데이터를 업데이트한 이후에도 일정 기간동안 이전 데이터를 보존할 수 있다. 이러한 특징을 응용하면 저장장치 수준에서 섀도우 페이징을 구현할 수 있으며, 특히 모바일 분야와 같이 섀도우 페이징을 구현하는데 부담이 큰 분야에서는 저장장치가 섀도우 페이징 기능을 지원하면 매우 큰 성능 향상을 기대할 수 있다. 본 논문에서는 NAND 플래시 기반 저장장치들의 특징을 활용하여 저장장치 수준에서 트랜잭션의 원소성을 보장하는 방안을 제시하고, 이를 통해 전체적인 저장 매체 성능이 향상될 수 있는 가능성에 대해 알아본다.

1. 서론

오늘날 NAND 플래시 메모리(이하 NAND) 기반 저장 장치들은 자기 디스크 기반의 HDD 에 비해 다양한 장점을 기지고 있다. 높은 랜덤 읽기/쓰기 성능, 낮은 전력 소모 및 발열, 충격에 대한 안정성 등을 예로 들 수 있는데, 이러한 특징들로 인해 점점 NAND 기반 저장장치가 다양한 분야에서 HDD 를 대체하는 저장장치로 인기를 얻고 있다. 또한, 특히 칩 형태로 초소형화 할 수 있어서, 모바일 분야에서는 NAND 기반 저장장치가 주요 저장매체로 사용되고 있다.

이러한 NAND 기반 저장장치들에는 특히 데이터베이스에서 매우 유용하게 활용될 수 있는 특징이 있다. 그것은 데이터가 업데이트된 이후에도 업데이트 되기 전 데이터가 일정 기간 동안 보존된다는 점이다. 이는 NAND 의 물리적인 특성으로 인해 생긴 독특한 저장 방식으로, 이 특징을 활용하면 DBMS 가 트랜잭션을 처리할 때 따로 데이터를 백업하지 않더라도 트랜잭션 실패를 처리할 수 있어 DBMS 의 트랜잭션 지원 부담을 상당히 덜어줄 수 있다.

저장장치가 트랜잭션의 기능을 분담하는 것은 특히 모바일 분야에서 큰 장점이 될 수 있다. 예를 들어 모바일 환경에서 폭넓게 활용되는 SQLite 의 경우 트랜잭션의 원소성을 보장하기 위해 저널링 기법을 사용하는데, SQLite 의 저널링은 모바일 시스템 전체의 성능에 큰 악영향을 미치고 있다[1, 2]. 따라서 만일 저장장치 수준에서 트랜잭션을 지원한다면 모바일 시스템 전체적인 성능 향상에 도움이 될 수 있다.

본 논문에서는 앞서 언급한 NAND 기반 저장장치의 특징을 활용하여 저장장치 수준에서 트랜잭션 처리를 지원하는 방법을 제안하고, 간단한 실험을 통해 이러한 시스템의 실현 가능성을 알아보고자 한다.

본 논문의 구성은 다음과 같다. 2장에서는 NAND 기반 저장장치들의 저장 방식 및 트랜잭션을 지원하기 위해 널리 쓰이는 섀도우 페이징에 대해 간단히 살펴보고, 저장매체에서 트랜잭션을 지원하는 방법에 대한 기존 연구들에 대해 알아본다. 그리고 3장에서는 본 논문에서 제안하는 방법론에 대해 살펴보고 4장에서는 실험을 통해 본 논문에서 제안하는 방법론의 실현 가능성을 살펴본다. 마지막으로 5장에서는 연구 결과에 대한 결론 및 향후 연구 방향을 제시한다.

2. 배경 지식

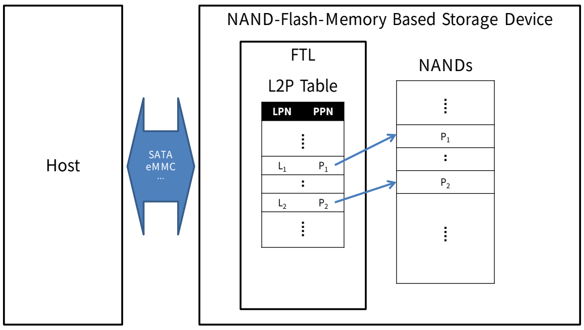
2.1. NAND 플래시 메모리에서의 Out-of-place 갱신 방식.

NAND 는 한 번 데이터를 쓰고(program) 나면 그 영역을 지우기(erase) 전까지는 덮어쓰기(over-write)가 불가능하다. NAND 기반 저장장치들은 이러한 특성 때문에 데이터를 업데이트 할 때 기존 데이터를 직접 수정하는 대신, 기존 데이터를 남겨두고 다른 빈 공간에 데이터를 옮겨 적고 주소에 대한 사상(mapping)만 변경하는(out-of-place) 업데이트 방식을 채택하고 있다. 따라서 호스트가 특정 주소에 있는 데이터에 대한 읽기/쓰기 동작을 요청하면, 호스트의 주소(논리

※ 이 논문은 2013 년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(2012R1A2A1A01010775).

* 교신저자

주소)를 NAND 의 물리 주소로 변환(translation)하는 과정을 거친다. NAND 기반 저장 장치들은 이러한 변환 과정을 처리하기 위해 플래시 변환 계층(Flash Translation Layer, 이하 FTL)을 두어 논리-물리 주소 사상(Logical-to-Physical Address Mapping, 이하 L2P)을 담당하게 하고 있다. (그림 1)은 이러한 FTL 의 구조를 개념적으로 나타낸 것이다.



(그림 1) NAND 기반 저장장치 구조

2.2. 섀도우 페이징[3]

섀도우 페이징은 데이터베이스 분야에서 트랜잭션의 ACID 특성을 보장하기 위해 쓰이는 방법 중의 하나이다. 이 방식은 트랜잭션이 진행되는 동안 트랜잭션에 의해 변경되는 데이터의 원본과 변경 이후를 모두 관리하다가 트랜잭션이 정상적으로 완료(커밋)될 경우 원본 데이터를 버리고 변경된 데이터를 적용하며, 반대로 트랜잭션 수행 도중 오류가 발생할 경우 변경된 데이터를 버리고 이전 데이터로 모두 되돌리는 방식이다.

모바일 분야에서 널리 쓰이는 SQLite 도 트랜잭션 처리하기 위해 섀도우 페이징 기법을 쓰고 있다.[4] SQLite 에서는 이전/이후 데이터를 서로 다른 파일로 나눠서 관리하다가 트랜잭션이 종료된 이후 두 파일을 합치는 방식을 사용하는데, 이 과정에서 부가적인 I/O 및 동기화(synchronization) 동작이 자주 발생하여 전체적인 시스템 성능에 악영향을 준다.[2]

2.3. 저장 장치 수준에서의 트랜잭션 지원

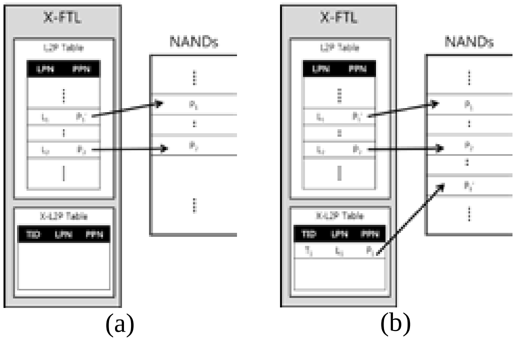
NAND 기반 저장 장치들이 사용하는 out-ouf-place 업데이트 방식은 트랜잭션 처리에 매우 유리하다. 즉, 적어도 일정 기간 동안 NAND 플래시 어딘가에 업데이트 이전/이후 데이터가 모두 남아 있으므로, 이전 데이터를 활용하면 DBMS 가 이전 데이터를 따로 백업하지 않고도 자연스럽게 섀도우 페이징 기법을 구현할 수 있다.

기존에도 NAND 기반 저장장치들의 이러한 특징을 활용하여 트랜잭션을 처리하는 다양한 방식들이 제안되어 왔다.[5-8] 특히 Ouyang et al.[6]이 제안하였던 방법은 실제로 FusionIO SSD 및 MySQL 드라이버에 적용되어 처리 성능을 크게 향상시켰다. 그러나, 이러한 방법론들은 트랜잭션 쓰기 순서에 제약[5]이 생기거나, 트랜잭션 커밋할 때 데이터 전체[6] 혹은 적어도 트랜잭션 전체의 데이터 쓰기 순서[8]를 전달해 줄 수 있어야 한다는 문제가 있다. 이러한 문제들로 인해 특히 가용 자원이 부족한 모바일 환경에서는 기존에 제안된 방법들을 쉽게 활용하기 어렵다.

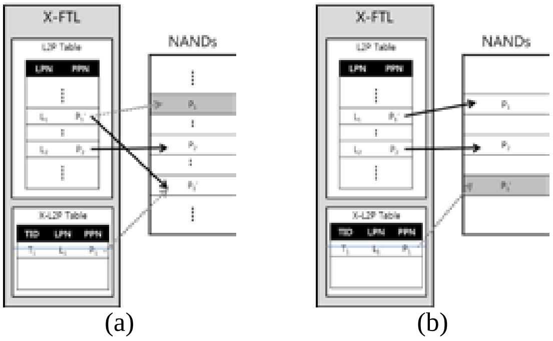
다만, 최근의 결과 중에서 Kang et al. 이 제안한 X-FTL[7]의 경우, 호스트 측면에서는 기존의 호스트 – 디바이스 간 인터페이스에 몇가지 간단한 기능만 추가하여 다른 방식들에 비해 모바일 환경에 좀 더 적합한 방식이라 할 수 있다.

X-FTL 는 일반적인 L2P 테이블에 덧붙여서 트랜잭션을 처리하기 위해 부가적인 L2P 테이블(X-L2P 테이블)을 추가하고, 트랜잭션을 처리할 때 X-L2P 테이블을 활용하는 방식이다. X-FTL 의 구체적인 트랜잭션 처리 방법 대해 살펴 보면 다음과 같다. 1) 우선 호스트로부터 트랜잭션 쓰기가 들어오면 L2P 테이블에 있는 기존 L2P(old-L2P) 정보는 유지한 상태로 트랜잭션 ID 및 새로 쓰여진 L2P 정보를 X-L2P 테이블에 등록한다.(그림 2-b) 이러한 작업을 트랜잭션 쓰기가 들어오는 동안 계속 반복하다가 3-1) 최종적으로 트랜잭션이 커밋될 경우 X-L2P 테이블에서 해당 트랜잭션과 연관된 모든 L2P 정보를 찾아서 L2P 테이블에 반영한다.(그림 3-b) 만일, 도중에 트랜잭션 실패가 발생할 경우에는 X-L2P 테이블의 내용을 버린다.(그림 3-b) 이와 같은 방식을 활용하면 DBMS 수준에서 특별한 동작 없이도 간단하게 커밋 롤백을 구현할 수 있다.

다만 X-FTL 의 방식은 몇 가지 단점이 있다.



(그림 2) X-FTL 업데이트 이전(a) - 이후(b): T_1 트랜잭션이 L_1 주소를 업데이트(P_1 à P_1')할 경우, L2P 테이블 그대로 두고 X-L2P 테이블에만 정보인 <T_1, L_1, P_1'>를 등록한다.



(그림 3) 트랜잭션이 커밋(a) 혹은 롤백(b)된 경우: (그림 2(b)) 상태에서 트랜잭션 T_1이 커밋될 경우 X-L2P 테이블의 모든 내용을 L2P 에 반영한다.(a) 만일 롤백될 경우에는 X-L2P 테이블의 정보만 지운다.(b)

우선, X-FTL이 지원할 수 있는 최대 트랜잭션 쓰기 횟수가 X-L2P 테이블의 크기에 제약을 받는다. 현재의 X-FTL은 최대 1000 개 페이지 [1]만큼의 트랜잭션 쓰기만 지원할 수 있으며, 더 많은 트랜잭션 쓰기를 지원하기 위해서는 더 많은 주 메모리 공간을 차지하게 된다. 그러나, 특히 메모리가 매우 작은 eMMC [2] 등의 환경에서는 X-L2P 테이블의 크기 [3]를 늘리는 데에는 한계가 있으며, 주 메모리 이외에 NAND에 나눠 저장하는 방법 등은 고려되지 않고 있다.

또, X-FTL 의 경우 트랜잭션을 수행하는 동안에는 전체 주소 공간에 대한 최신 L2P 정보가 L2P 테이블과 X-L2P 테이블에 분산되어 있다. 호스트부터 R/W 접근이 발생하면 FTL 이 최신 L2P 정보를 찾게 되는데, X-FTL 의 경우 트랜잭션을 수행하는 동안 R/W 접근이 들어오면 최신 L2P 정보를 얻기 위해 원래의 L2P 테이블뿐만 아니라 X-L2P 테이블도 같이 검색해 보아야 한다. 특히, X-L2P 크기가 매우 커서 X-L2P 를 주 메모리 안에 다 수용할 수 없을 경우 L2P 정보를 얻는데 드는 비용(부담, overhead)이 문제가 될 수 있다.

마지막으로, X-FTL 의 경우 트랜잭션이 커밋될 때 X-L2P 테이블에 있는 L2P 정보를 한꺼번에 반영하므로 커밋 동작이 복잡하다는 단점이 있다. 대신 트랜잭션이 커밋되기 전에는 L2P 테이블이 트랜잭션의 영향을 받지 않고, 따라서 트랜잭션이 실패할 경우에도 롤-백 과정이 매우 간단해진다는 장점이 있지만, 일반적인 상황이라면 트랜잭션이 실패할 확률보다 정상적으로 커밋되는 확률이 더 클 것이므로, 롤-백 대신 커밋 과정이 복잡하다는 것은 잠재적인 부담(overhead)로 작용할 수 있다.

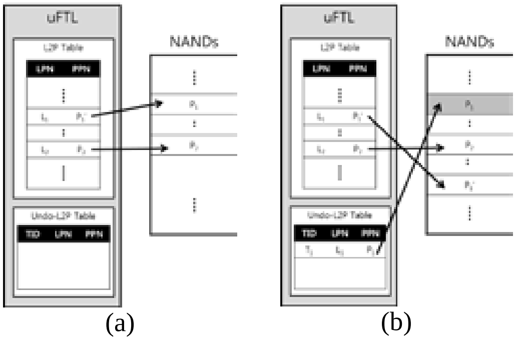
3. 제안하는 방법

본 논문에서는 X-FTL[7]을 기반으로 하여, 성능 향상을 위해 X-L2P 테이블 대신 Undo-L2P 테이블을 두는 FTL(undo-FTL, 이하 uFTL)을 제안한다.

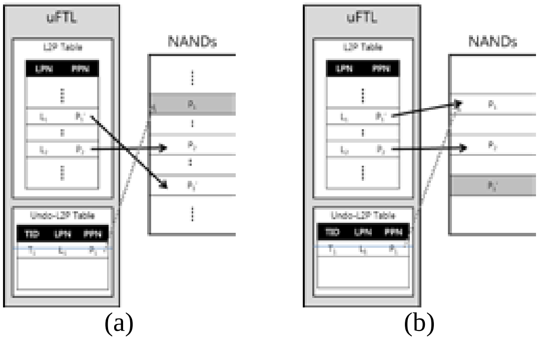
uFTL 의 기본적인 동작 방식은 다음과 같다:

1) 호스트로부터 트랜잭션 쓰기가 들어올 경우, 데이터를 빈 영역에 쓰고, L2P 테이블에 Undo-L2P 테이블에 추가한다.((그림 4 참조)
2) Undo-L2P 테이블에 있는 정보들은 트랜잭션이 커밋 혹은 롤백되기 전까지 계속 유지된다. 만일 트랜잭션이 커밋될 경우에는 해당 트랜잭션과 연관된 Undo-L2P 정보들은 모두 무효화되며, 트랜잭션이 중단되거나 정상적으로 종료되지 않을 경우에는 Undo-L2P 테이블의 정보를 활용하여 L2P 정보를 복구한다.((그림 5참조)

NAND 기반 저장장치가 위와 같이 동작하기 위해서는 우선 호스트 – 디바이스 간의 인터페이스에 아래와 같이 몇 가지 추가 기능이 구현되어야 한다.



(그림 4) uFTL 업데이트 이전(a) - 이후(b): T_1 트랜잭션이 L_1 주소를 업데이트(P_1 à P_1')할 경우, L2P 테이블은 최신 정보로 업데이트(<L_1, P_1> à <L_1, P_1'>되고, Undo-L2P 에 기존 정보인 <T_1, L_1, P_1>를 등록한다.



(그림 5) 트랜잭션이 커밋(a) 혹은 롤백(b)된 경우: (그림 **4**-b) 상태에서 트랜잭션 T_1이 커밋될 경우 Undo-L2P 테이블의 정보만 지워진다.(a) 만일 롤백될 경우 L2P 테이블의 정보를 Undo-L2P 테이블 정보대로 복구한다.(b)

- **Write(tid, lba)**트랜잭션에 포함되는 쓰기/업데이트일 경우, 주소 및 섹터 카운트 정보 뿐만 아니라 현재의 쓰기 동작이 어떤 트랜잭션에 속하는지를 구분하기 위해 트랜잭션 ID 정보를 디바이스로 전달한다.
- **Commit(tid)** 특정 트랜잭션(tid)이 정상적으로 커밋 된 경우
- **Abort(tid)** 트랜잭션 처리 도중 오류 등이 발생하여 정상적으로 커밋되지 못하고 원상복귀(roll-back)되어야 할 경우

위의 3 가지 기능들은 기존의 SATA, eMMC 등의 인터페이스에는 없는 새로운 기능들이지만, 기존의 X-FTL[7]와 비교할 때 거의 비슷한 수준의 확장이며, 읽기 시에는 트랜잭션 ID 를 고려하지 않아도 되므로 X-FTL 에 비해 수정 범위가 적다.

uFTL 을 기존의 X-FTL 과 비교해 보면, 최신 L2P 정보가 L2P 와 X-L2P 에 나눠서 저장되는 X-FTL 과는 달리 uFTL 에서는 최신 L2P 정보가 항상 L2P 테이블에만 존재하게 된다. 따라서, 최신 L2P 정보를 찾을 때에도 L2P 테이블 검색 이외의 다른 추가 비용이 발생하지 않으며, 되돌리기-L2P 테이블의 크기나 위치에 따른 추가 비용 없이 L2P 변환을 처리할 수 있다는 장점이 있다.

또한, 트랜잭션이 커밋 될 때 X-L2P 정보를 전부

[1] 페이지당 4kB 일 경우, 최대 4MB

[2] 보통 주 메모리가 SRAM 으로 수십 ~ 수백 kB 수준이며, 대부분의 경우 L2P 테이블 등이 사용하고 있고, 주 메모리 크기가 비용과 직결되므로 여유공간이 거의 없이 활용하는 것으로 알려져 있다.

[3] 현재 X-FTL 의 경우 X-L2P 테이블이 1000 개 기준으로 약 16kB 정도 차지한다.

L2P 에 반영해야 하는 X-FTL 과 달리, uFTL 은 이미 L2P 테이블에 새로운 정보가 반영되어 있으므로 Undo-L2P 테이블에서 현재 트랜잭션 정보를 지우는 것 이외에는 부가적인 처리가 필요하지 않다는 장점도 있다.

4. 실험

본 논문에서 제안하는 아이디어의 구현 가능성 및 타당성을 검증하기 위해 두 개의 OpenSSD[9] 에뮬레이터를 구현하였다. 하나는 OpenSSD-1.1.0 펌웨어[10]의 동작 방식을 그대로 따르는 에뮬레이터이고, 나머지 하나는 앞의 에뮬레이터에 본 논문의 아이디어인 Undo-L2P 를 더하여 uFTL 을 구현한 FTL 이다. 두 에뮬레이터는 모두 호스트의 I/O 트레이스를 받아서 이를 수행하고, NAND 에서 발생하는 읽기/쓰기/지우기 동작의 횟수를 측정할 수 있다.

4.1. 실험 방법

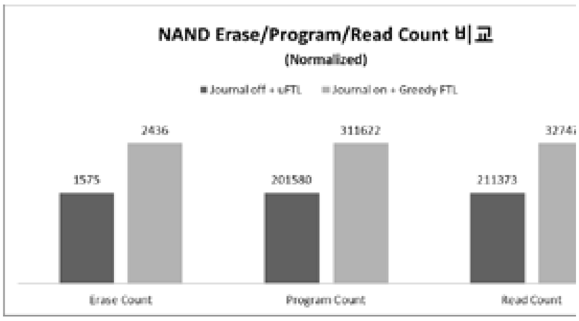
실험에서는 uFTL 이 적용되었을 때 SQLite 의 성능 개선 정도를 알아보기 위해, 1) 저널링 모드를 끈 상태에서 uFTL 의 트랜잭션 처리 지원을 받는 경우와 2) uFTL 이 아닌 일반 FTL 을 사용하면서 SQLite 의 저널링 기능을 사용하는 경우를 비교하였다.

실험을 위해서 우선 리눅스 환경에서 TPC-C 와 유사한 형태의 쿼리[11] 10000 회를 SQLite 의 저널 모드를 켠 상태와 끈 상태에서 각각 수행하고, 이 때 데이터베이스 파일이 저장되어 있는 물리 장치에 대한 R/W 트레이스 및 트랜잭션 정보를 추출하였다.

추출한 트레이스들 중 우선 1)에 대한 성능을 실험하기 위해 uFTL 에뮬레이터에서 SQLite 저널모드를 끈 트레이스를 수행하였고, 2)의 실험을 위해 일반적인 OpenSSD 펌웨어에서 SQLite 의 저널 모드를 켠 트레이스를 수행하였다.

4.2. 결과 및 분석

실험을 통해 앞의 1), 2)의 상황에 대해 가상 NAND 상에서 발생하는 지우기(Erase), 쓰기(Program), 및 읽기(Read) 동작의 횟수를 측정하였으며, 이를 그림으로 나타내면 아래 (그림 6)과 같다.



(그림 6) SSD 시뮬레이션 결과

실험 결과, 1)의 경우 2)의 경우보다 세 가지 동작 모두 모두 약 35% 정도 적게 발생하는 것을 확인할 수 있었다. 이러한 결과는 2)의 경우에는 SQLite 가 자체적으로 섀도우 페이징을 수행하면서 상당한 오버헤드를 발생키는데 비해, 1)의 경우에는 uFTL 자체적으로 트랜잭션을 처리하기 때문에 그러한 오버헤드가 발생하지 않기 때문이다.

5. 결론 및 향후연구

지금까지 저장장치 수준에서 트랜잭션을 지원하는 방법으로서 FTL 에 되돌리기 테이블을 사용하는 방법에 대해 제시하고, 실험을 통해 가능성을 간략하게 살펴보았다. 제시한 되돌리기 테이블 방식은 호스트와의 인터페이스에 몇 가지 기능만 추가하면 구현 가능하므로 다른 방식들에 비해 비교적 간단하게 구현할 수 있으며, 트랜잭션 처리를 위한 섀도우 페이징을 호스트 쪽에 구현하면서 생기는 성능 상의 부담을 줄여줄 수 있다.

향후에는 제시한 아이디어를 실제 OpenSSD[9]에 구현하여 실제 성능 및 수행시간을 직접 비교해 보고, 좀 더 다양한 워크로드에서 실험을 수행할 계획이다. 또한, X-FTL[7] 등 다른 방법들과의 비교 실험도 진행할 것이다.

참고문헌

[1] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *Trans. Storage*, vol. 8, pp. 1-25, 2012.

[2] K. Lee and Y. Won, "Smart layers and dumb result: IO characterization of an android-based smartphone," presented at the ACM International Conference on Embedded Software, Tampere, Finland, 2012.

[3] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 6th ed.: McGraw-Hill, 2011.

[4] SQLite. *Atomic Commit In SQLite*. Available: http://www.sqlite.org/atomiccommit.html

[5] P. Sunhwa, Y. Ji Hyun, and O. Seong-Yong, "Atomic write FTL for robust flash file system," in *Consumer Electronics, 2005. (ISCE 2005). Proceedings of the Ninth International Symposium on*, 2005, pp. 155-160.

[6] O. Xiangyong, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block I/O: Rethinking traditional storage primitives," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011, pp. 301-311.

[7] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, "X-FTL: transactional FTL for SQLite databases," presented at the ACM SIGMOD International Conference on Management of Data, New York, New York, USA, 2013.

[8] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," presented at the 8th USENIX conference on Operating systems design and implementation, San Diego, California, 2008.

[9] *The OpenSSD Project*. Available: http://www.openssd-project.org/wiki/The_OpenSSD_Project

[10] *OpenSSD-1.1.0 Firmware*. Available: http://www.openssd-project.org/wiki/Downloads

[11] A. Pavlo. (2011). *pytpcc*. Available: https://github.com/apavlo/py-tpcc/blob/master/pytpcc/tpcc.py