JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 27 No. 12, pp. 1-8, December 2022
https://doi.org/10.9708/jksci.2022.27.12.001

An Improved Index Structure for the Flash Memory Based F2FS File System

Yong-Seok Kim*

*Professor, Dept. of Computer Engineering, Kangwon National University, Gangwon-do, Korea

[Abstract]

As an efficient file system for SSD(Solid State Drive), F2FS is employed in the kernel of Linux operating system. F2FS applies various methods to improve performance by reflecting the characteristics of flash memory. One of them is improvement of the index structure that contains addresses of data blocks for each file. This paper presents a method for further improving performance by modifying the index structure of F2FS. F2FS manages all index blocks as logical numbers, and an address mapping table is used to find the physical block addresses of index blocks on flash memory. This paper shows performance improvement by applying logical numbers to the last level index blocks only. The count of mapping table search for a data block access is reduced to 1~2 from 1~4.

▸ **Key words**: F2FS, File System, Index Structure, SSD, Flash Memory

[요 약]

F2FS는 SSD(Solid State Drive)를 위한 파일시스템 중의 하나로서 리눅스 운영체제의 커널에 채용되어 널리 사용되고 있다. F2FS는 플래시 메모리의 특성을 반영하여 성능을 높이기 위한 여러 가지 방안들을 적용하였는데, 그 중의 하나가 파일별 데이터 블록들의 주소 정보를 관리하는 인덱스 구조의 개선이다. 본 논문에서는 F2FS의 인덱스 구조를 더욱 개선하여 성능을 높이는 방안을 제시하였다. F2FS는 모든 인덱스 블록들에 대하여 논리적 번호로 기록하고 이것을 물리적 번호로 매핑하는 테이블을 사용한다. 본 논문에서는 인덱스 블록들 중에서 끝단의 블록만 논리적 번호를 적용하고 앞단의 블록들은 물리적 번호를 직접 적용함으로써, 데이터 블록 접근시에 매핑 테이블을 검색하는 회수를 기존의 1~4회에서 1~2회로 줄일 수 있음을 보여주었다.

▸ **주제어**: F2FS, 파일 시스템, 인덱스 구조, SSD, 플래시 메모리

• First Author: Yong-Seok Kim, Corresponding Author: Yong-Seok Kim
*Yong-Seok Kim (yskim@kangwon.ac.kr), Dept. of Computer Engineering, Kangwon National University
• Received: 2022. 09. 22, Revised: 2022. 12. 12, Accepted: 2022. 12. 12.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

반도체 기술의 발전으로 플래시 메모리 기반의 저장장치인 SSD(Solid State Drive)는 기존의 HDD (Hard Disk Drive)를 급속하게 대체해가고 있다. 성능 측면의 장점에다가 비교적 열세였던 가격마저도 충분히 경쟁력을 갖게 되었기 때문이다. 초기에는 저전력 및 소형화가 중요한 이동형 정보기기들 위주로 채용되기 시작했지만 이제는 대규모 서버 시스템들에도 주류로 자리를 잡았으며 다양한 제품들이 상품화되어 있다.

플래시 메모리는 하드디스크에 비해서 여러 가지 장점들이 있지만 결정적으로 제자리 덮어쓰기가 허용되지 않는 문제가 있고, 게다가 최근에 비트당 용량을 늘리는 기술들이 채용되면서 순차쓰기만 허용되는 제한이 있다. 이러한 문제들을 극복하여 SSD를 대규모 저장장치로 사용하는 방법들이 다양하게 제안되어 사용되고 있다. 대표적인 것들 중 하나는 SSD가 기존의 HDD와 동일한 인터페이스를 제공하는 것이다. 그러면 기존에 HDD 기반으로 개발된 다양한 파일시스템들을 그대로 사용할 수 있다. 그러기 위해서 SSD 내부에서는 자체 컨트롤러를 활용하여 플래시 변환 계층(Flash Translation Layer)에서 호스트로부터 요청된 블록을 적절한 위치에 배치하여 기록하고 호스트로부터의 논리적 블록 번호를 내부의 물리적 블록 번호로 매핑하는 방법을 사용한다[1].

다른 방법으로는 SSD의 특성을 반영하여 완전히 새로운 파일시스템을 개발하는 것이다. 그러한 대표적인 예로는 JFFS3(Journalling Flash File System 3)[2], F2FS(Flash-Friendly File System)[3] 등이 있다. 로그 구조 파일 시스템 (LFS: Log-structured File System)[4]은 파일에 대한 임의위치 쓰기를 순차쓰기로 대체하는데, 이것은 순차쓰기만 허용하는 플래시 메모리를 적용하는 SSD에서 특히 유용하다. JFFS3[2], NILFS2(New Implementation of a Log-structured File System)[5], BTRFS(B-Tree File System)[6] 등이 LFS 기반으로 개발된 것이다. 비교적 최근에 개발되어 리눅스 커널 공식 배포 버전에 포함된 F2FS는 리눅스에서 사용하는 대표적인 파일 시스템인 ext4에 근거를 두지만 플래시 메모리 특성을 반영하여 완전히 새롭게 작성되었으며, 기존의 LFS 기반의 파일시스템 들에 비해서 우수한 성능을 보여주었다[3].

F2FS는 삼성전자에서 완전히 새롭게 개발하여 공개한 것인데, 이름의 'Flash-Friendly'라는 표현에서 보듯이 플래시 메모리의 특성을 반영한 파일 시스템으로서 성능 향상을 위하여 여러 가지 방안들이 적용되어 있다. 효율적인 인덱스 구조, 플래시 메모리 친화적인 레이아웃 배치, 다중 헤드 로깅, 적응적 로깅, 롤-포워드 복구 등이 포함된다.

리눅스 운영체제는 F2FS를 커널 3.8에 처음 도입한 이래로 플래시 메모리용 주요 파일 시스템중의 하나로서 지속적으로 관리하고 있다[7]. 또한 스마트 폰을 비롯한 여러 상용 제품들에도 공식적으로 채택되고 있으며 기존의 ext4 파일 시스템에 비해서 성능이 향상되었다고 보고되고 있다[8].

F2FS의 성능향상을 위한 여러 가지 시도들이 있었다. L. Yang 등은 F2FS 파일시스템에서 파일의 데이터가 흩어져서 기록되는 단편화에 의해서 성능이 저하되는 문제를 개선하는 방안을 제시하였다[9]. 파일별로 기록 특성이 다른 점을 활용하여, 재기록이 잦은 일부 파일을 선택하여 연속된 플래시 메모리 영역을 미리 확보하는 방안을 적용하였다. Y. Lee 등은 SSD의 디바이스 내부에 주소 매핑 정보 수정 기능을 추가함으로써 파일시스템 차원에서는 제자리 기록이 가능하도록 할 수 있고 따라서 세그먼트 정리 오버헤드와 파일별 메타 데이터 수정 오버헤드를 줄일 수 있음을 보여주었다[10].

본 논문에서는 F2FS의 인덱스 구조에 초점을 맞추어서 그 성능을 개선하는 방안을 기술한다. F2FS가 기존의 LFS 기반의 파일시스템들보다 인덱스 구조를 개선하였는데, 본 논문에서는 이것을 더욱 발전시키는 것이다. 그동안의 F2FS에 대한 개선 제안들은 F2FS의 인덱스 구조를 그대로 유지하는 것을 전제로 한 것이므로 본 논문은 이들 모두에게도 그대로 적용될 수 있다.

II. Related Works

1. Index Structure of ext4 File System

리눅스의 대표적인 파일시스템인 ext4는 파일별로 inode를 할당하여 여러 가지 메타 데이터를 기록한다. 메타 데이터에는 파일의 크기, 소유자, 수정시간, 보호모드 등이 포함된다. 파일의 특정 오프셋에 해당하는 데이터 블록의 위치는 인덱스 정보로 기록한다. 현재의 ext4 inode 구조에는 15개의 블록 번호가 기록되는데 그림 1과 같이 구성된다, 앞의 12개의 직접 블록들(Direct Blocks)은 직접 데이터 블록들의 번호를 기록한다. 12개 블록 크기 이내의 파일들은 이 정보만으로 충분하다. 파일의 크기가 그 이상으로 커지면 간접 블록들을 활용한다.

13번째의 1단 간접블록(Single Indirect Block)은 인덱스 블록(Level 1 Index Block)의 주소인데, 인덱스 블록에는 데이터 블록들의 주소가 배열 형태로 기록된다. 예를 들어서 한 블록의 크기가 4K 바이트이고 블록 주소는 32비트(4바이트)로 표시된다면 인덱스 블록에는 1024개의 데이터 블록 주소들이 기록된다. 파일 크기가 커짐에 따라서 14번째의 2단 간접블록(Double Indirect Block)은 2단계(Level 2 Index Block)까지, 15번째의 3단 간접블록(Triple Indirect Block)은 3단계(Level 3 Index Block)까지 확장된다. 따라서 이 구조로 기록할 수 있는 파일의 최대 크기는 하나의 인덱스 블록에 1024개의 블록 주소를 기록한다고 가정할 때 1024^3개의 4K 바이트 크기 데이터 블록들, 즉 4T 바이트 크기까지 가능하므로 현존하는 저장장치들을 다 수용할 수 있게 된다.

ext4 파일 시스템은 HDD에 적용하기 위해서 개발되었는데, SSD에서는 플래시 메모리의 특성상 제자리 덮어쓰기가 허용되지 않으므로 그대로 사용할 수가 없다. 그러나 내부에 FTL(Flash Translation Layer) 기능을 가진 SSD를 사용하면, 파일 시스템에서 요청한 논리적 블록 번호를 디바이스 차원에서 실제로 기록된 블록 번호로 매핑해서 처리하므로 ext4를 그대로 사용할 수 있게 된다.

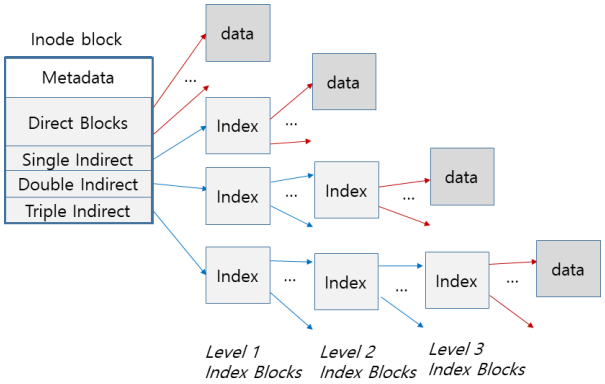

Fig. 1. ext4 Index Structure

2. Index Structure of Log-structured File Systems

LFS 기반의 파일시스템들은 inode 및 인덱스 블록들에 대상 블록의 물리적 주소를 기대로 기록한다. 그 결과 데이터 블록의 읽기에는 아무 문제가 없지만, 데이터 블록의 쓰기에는 많은 오버헤드가 수반된다. 플래시 메모리는 제자리에 덮어쓰기가 되지 않으므로 특정 데이터 블록을 새로운 데이터로 기록하면 블록 주소가 변경되므로 그에 따른 인덱스 블록의 정보도 수정해야 한다. 직접 블록일 경우에는 inode만 수정하면 된다. 그러나 간접 블록들은 더욱 복잡해진다. 연관된 인덱스 블록들도 모두 수정해야 하기 때문이다.

흔히 '헤매는 트리'(wandering tree) 문제라고 하는데[2], 최악의 경우인 3단 간접 블록에 해당할 경우에는 4번에 걸쳐서 인덱스 정보들을 차례대로 수정해야 한다. 데이터 블록에 쓰기를 하면 기록된 주소가 바뀌었기 때문에 레벨 3 인덱스 블록의 정보를 수정해야 하고, 이 과정에서 레벨 3 인덱스 블록의 주소가 바뀌었기 때문에 레벨 2 인덱스 블록을 수정해야 하는 식의 과정이 줄줄이 이어진다. 마지막에 inode까지 수정해야 한다. 이러한 과정에 의해서 '쓰기 증폭'(write amplification)의 효과가 심하게 발생한다. 데이터 블록을 한번 수정하기 위해서 추가로 4번의 쓰기가 수반된 것이다. 이러한 과정은 제자리 덮어쓰기 뿐만 아니라 파일의 크기가 커지면서 데이터 블록을 추가하여 기록할 때도 마찬가지다.

데이터 블록의 쓰기 과정에는 쓰기 증폭 문제가 심각하게 발생하지만, 데이터 블록의 읽기 과정에는 추가 오버헤드가 없다. inode 정보로부터 시작하여 인덱스 블록들의 주소를 차례대로 알 수 있기 때문이다. 물론 inode를 읽기 위해서는 inode 테이블을 검사해야 하는 오버헤드가 있지만 이것은 다른 플래시 메모리 기반의 파일시스템들과 마찬가지로 불가피하다.

inode들은 고유한 번호를 가지고 있고 그 번호를 통하여 inode가 저장된 블록 주소를 알아내야 하므로 inode 테이블을 적용한다. 따라서 inode 정보를 읽기 위해서는 먼저 inode 테이블을 검사하는 과정을 거쳐야 하는데, 이 테이블의 정보도 기본적으로 플래시 메모리에 기록되어야 한다. inode 정보가 수정되면 그 변경된 블록 주소에 따라서 inode 테이블도 수정되어야 하며, 따라서 플래시 메모리에 쓰기 작업이 추가된다. 이 부분에 대한 오버헤드는 3.2절의 끝 부분에서 별도로 분석하도록 한다.

3. Index Structure of F2FS File System

리눅스 운영체제에서 SSD를 위해 사용되는 표준 파일 시스템 중의 하나가 F2FS이다. F2FS는 기존의 LFS 기반의 파일시스템을 개선하기 위해서 여러 가지 방안들을 적용하였는데, 그 중에서 본 논문은 인덱스 구조에 집중하여 설명한다. LFS 기반의 파일 시스템에서 발생하는 쓰기 증폭 문제를 개선하기 위한 방안으로서 인덱스 구조를 그림 2와 같은 구조로 개발하였다.

LFS에서는 inode 및 인덱스 블록들에 그 다음 단계의 인덱스 블록들의 물리적 주소(PA: Physical Address)를 그대로 기록하는 데 반해서, F2FS에서는 모든 인덱스 블록들에 논리적인 번호(LN: Logical Number)를 부여하고

그 번호를 기록한다. 끝단에 해당하는 인덱스 블록에는 데이터 블록의 물리적 주소(PA)를 그대로 기록한다. 그림 2에서 보는바와 같이 직접 블록들은 inode에 데이터 블록들의 PA를 그대로 기록한다. 간접 블록의 경우에 레벨 1 인덱스 블록의 논리적 번호(LN)를 기록한다. 인덱스 블록에는 다음 단계도 인덱스 블록일 경우에는 그 LN을 기록하고, 데이터 블록일 경우에는 그 PA를 기록한다.

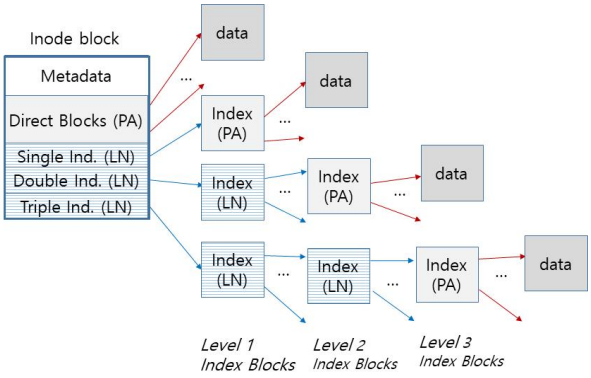

Fig. 2. F2FS Index Structure

이 방법을 적용하면 쓰기 증폭 문제가 완화되는 근거는 다음과 같다. 특정 데이터 블록에 쓰기가 이루어지면 그 PA를 끝단의 인덱스 블록에 기록한다. 이 때 인덱스 블록은 새로운 주소에 기록되지만 그 LN은 변함이 없으므로 상단의 인덱스 블록을 수정할 필요가 없게 된다. 데이터 블록에 쓰기가 이루어지면 끝단의 인덱스 블록만 수정하면 되는 것이다. 따라서 LFS 기반의 파일시스템에서 발생하는 '헤메는 트리'문제가 발생하지 않는다. 직접 블록들의 경우에는 inode를 수정한다.

F2FS에서는 inode들 뿐만 아니라 모든 인덱스 블록들에 대해서도 실제로 기록된 물리적 블록 번호를 매핑하는 테이블이 필요하다. NAT(Node Address Table)가 그러한 목적으로 사용되는데[3], LFS 기반의 파일 시스템에서 사용하는 inode 테이블을 확장하여 인덱스 블록들까지 포함하도록 확장한 것으로 보면 된다. NAT를 관리하는 과정에는 별도의 오버헤드가 발생하는데 3.2절의 끝 부분에서 LFS의 경우와 함께 분석하도록 한다.

III. The Proposed Scheme

1. Improvement of F2FS Index Structure

본 논문은 F2FS의 인덱스 구조를 개선하여 성능을 높이는 것이 목적이다. 우선 데이터 블록을 접근할 때 그 물리적 블록 주소를 결정하는 과정을 살펴보자. LFS 기반의 파일 시스템에서는 inode 정보를 읽으면 차례대로 다음 단계의 인덱스 블록의 물리적 주소를 바로 알 수 있다. 직접 블록은 inode 정보로부터 바로 알 수 있고, 단계가 제일 긴 3단 간접 블록의 경우에는 레벨 1, 레벨 2, 레벨 3의 인덱스 블록들을 차례대로 읽어내면 목적하는 데이터 블록의 물리적 주소를 확인할 수 있다.

반면에 F2FS는 간접 블록의 경우에 NAT를 검사해야 하는 오버헤드가 추가된다. 직접 블록의 경우에는 inode 정보로부터 데이터 블록의 물리적 주소를 바로 알 수 있지만, 간접 블록의 경우에는 인덱스 블록을 읽을 때 마다 먼저 NAT를 검사하여 LN으로부터 PA를 알아내야 한다. 3단 간접 블록의 경우에는 인덱스 블록들을 읽기 위해서 3번에 걸쳐서 NAT 검사를 해야 한다.

본 논문에서 제시하는 인덱스 구조는 데이터 블록의 주소를 결정하는 과정의 오버헤드를 F2FS보다 줄이는 것이 목적이다. 그 핵심은 NAT를 검사하는 회수를 단축하는 것이다. 그림 3과 같이 인덱스의 끝 단과 바로 앞단의 인덱스 블록은 F2FS와 동일하게 관리한다. 그러나 그 앞단부터는 인덱스 블록의 LN이 아니라 해당 블록이 기록된 PA를 그대로 기록한다. 결과적으로 직접 블록들과 1단 간접 블록은 F2FS와 동일하다. 그러나 2단 간접 블록은 inode에 레벨 1 인덱스 블록의 PA를 기록한다. 3단 간접 블록은 inode와 레벨 1 인덱스 블록에는 다음 단의 인덱스 블록의 PA를 기록한다.

그림 4는 특정 파일의 특정 오프셋 부분에 대한 데이터 블록의 물리적 주소를 결정하는 과정을 보여준다. 파일의 inode 번호로부터 inode가 기록되어 있는 블록의 물리적 주소를 얻기 위해서 NAT를 한번 검사한다. 오프셋이 직접 블록에 해당하면 읽어낸 inode 정보로부터 바로 데이터 블록의 물리적 주소를 결정할 수 있다. 오프셋의 위치가 간접 블록에 해당하면 NAT 검사는 한번으로 충분하다. 1단 간접블록들은 inode에 기록된 LN으로부터 NAT를 검사하여 레벨 1 인덱스 블록의 PA를 알아야 하므로 총 2회의 NAT 검사를 하게 된다. 2단 간접 블록은 inode에 레벨 1 인덱스 블록의 PA가 기록되어 있으므로 NAT의 검사가 필요 없고, 그기에 기록된 레벨 2 인덱스 블록의 LN으로부터 NAT를 검사하여 그 PA를 확인하게 된다. 따라서 총 2회의 NAT 검사를 하게 된다. 마찬가지로 3단 간접블록도 inode를 위한 검사와 레벨 3 인덱스 블록의 PA를 확인하기 위한 검사를 하므로 총 2회의 NAT 검사만 필요하다. 만약에 미래에 더 확장을 해서 4단, 5단의 간접 블록들을 사용하게 되더라도 NAT의 검사회수는 2회 이내로 제한된다.

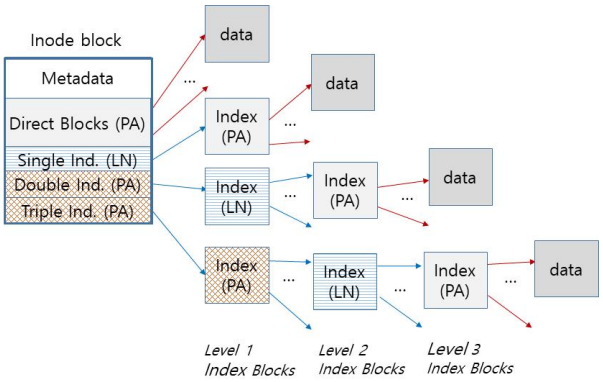

Fig. 3. Proposed Index Structure

```
Let IN be the inode number of the target file.
Let OFFSET be the target file offset.
BA = NAT_Search(IN)
INODE = ReadFlashBlock(BA)
if OFFSET is in a direct block
    return the PA of OFFSET on INODE
if OFFSET is in a single indirect block
    LN = the LN of OFFSET on INODE
else        // double or triple indirect block
    BA = the PA of OFFSET on INODE
    INDEX = ReadFlashBlock(BA)
    if OFFSET is in a triple indirect block
        BA = the PA of OFFSET on INDEX
        INDEX = ReadFlashBlock(BA)
    LN = the LN of OFFSET on INDEX
BA = NAT_Search(LN)
INDEX = ReadFlashBlock(BA);
return the PA of OFFSET on INDEX
```

Fig. 4. Determining Physical Address of a Data Block

개선된 인덱스 구조를 적용하면 NAT를 검사하는 오버헤드는 F2FS에 비해서 개선하면서도 쓰기 증폭 문제의 완화는 F2FS 수준을 달성한다. 우선 쓰기 증폭 문제에 대해서 가장 단계가 긴 3단 간접 블록을 대상으로 검토해 보면 다음과 같다. 데이터 블록에 쓰기가 이루어지면 변경된 PA를 레벨 3 인덱스 블록에 수정한다. 레벨 3 인덱스 블록이 수정되었지만 레벨 2 인덱스 블록에는 그 LN을 기록하고 있으므로 수정할 필요가 없다. 당연히 그 상단의 인덱스 블록들도 수정할 필요가 없게 된다. 따라서 쓰기 증폭 문제에 대해서는 F2FS와 마찬가지로 한 번의 인덱스 블록 수정으로 완료된다.

본 논문의 성능 개선은 데이터 블록을 접근할 때 그 PA를 결정하는 과정에서 발휘된다. NAT에서 해당 정보가 기록된 블록의 주소를 확인하는 검사 회수가 줄어들기 때문이다. 직접 데이터 블록의 경우에는 inode 정보로부터 데이터 블록의 위치를 바로 알 수 있기 때문에 차이가 없지만, 간접 단계가 늘어날수록 성능 개선 효과가 커진다. 3단 간접 블록의 경우를 보면, inode 정보로부터 바로 레벨 1 인덱스 블록의 PA를 알 수 있고, 레벨 1 인덱스 블록을 읽으면 바로 레벨 2 인덱스 블록의 PA를 알 수 있다. F2FS에서는 매번 NAT를 검사해야 했지만 그러한 과정이 생략되는 것이다.

2. Performance Analysis

본 논문에서 제안한 개선된 인덱스 구조의 성능 향상을 확인하기 위해서 F2FS 및 LFS 기반의 파일 시스템과 비교한다. 성능의 향상은 데이터 블록 접근시에 NAT를 검사해야 하는 회수를 줄임으로써 이루어지는데, 비교하면 표 1과 같다. NAT를 검사하는 회수의 감소가 전체 파일시스템의 성능에 미치는 영향은 이 절의 끝부분에서 추가로 언급한다.

Table 1. Number of NAT Search on a Data Block Access

Index Structure	Proposed	F2FS	LFS
Direct Block	1	1	1
Indrect Block	2	2	1
Double Ind. Block	2	3	1
Triple Ind. Block	2	4	1

먼저 inode를 읽기 위해서 노드 번호로부터 실제 기록된 PA를 알아야 하므로 세 가지 모두 NAT를 한번 검사해야 한다. LFS는 inode 정보로부터 인덱스 블록들의 PA를 직접 알 수 있으므로 더 이상의 NAT 검사는 없다. F2FS는 직접 블록의 경우에는 바로 데이터 블록의 PA를 알 수 있지만 단계가 늘어날수록 매번 NAT를 검사하여 인덱스 블록의 LN으로부터 그 PA를 알아내야 한다. 간접 단계가 늘어날수록 NAT 검사회수가 늘어나는 것이다.

본 논문에서 제안한 개선된 방식은 간접 단계가 늘어나더라도 NAT 검사회수는 inode의 검사를 포함하여 최대 2회로 제한된다. 그림 5는 파일의 크기에 따라서 NAT를 검사하는 회수를 시뮬레이션을 통하여 F2FS와 비교하였다. 파일을 랜덤 액세스한다고 가정하고 임의로 선택된 오프셋에 대하여 NAT를 검사하는 회수를 시뮬레이션으로 얻은 결과이다. 수치는 1천회 액세스에 대한 평균값이다. 파일 크기별로 추세를 보기 위해서 파일의 크기를 2배씩 증가시켜가면서 시뮬레이션을 진행하였다. F2FS는 파일의 크기가 증가할수록 NAT 검사회수가 증가하지만 본 논문에서 제안한 방식은 최대 2회로 제한되는 것을 보여주고 있다.

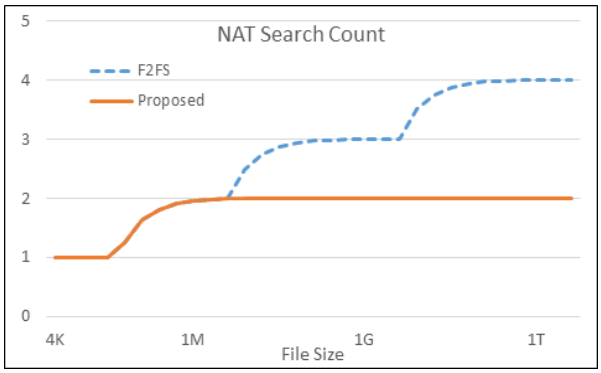

Fig. 5. Comparison of NAT Search Count

파일의 데이터 블록 읽기 작업이라면 그림 4의 과정을 거쳐서 확인한 데이터 블록의 PA를 적용하여 읽으면 그것으로 완성된다. 하나의 데이터 블록을 읽기 위해서는 inode 및 관련된 인덱스 블록들을 차례대로 읽어야 하는데, 그 과정에서 NAT를 검사하는 오버헤드가 수반된다. 개선된 인덱스 구조를 적용하면 그러한 오버헤드를 줄이게 되는 것이다.

파일의 데이터 블록 쓰기 작업은 '쓰기 증폭' 문제가 수반되는데, 표 2는 그 과정의 오버헤드를 비교하여 보여준다. 간접 단계가 늘어날수록 오버헤드도 늘어나는데 가장 긴 3단 간접 블록에 대하여 비교하였다. 먼저 쓰기 대상인 데이터 블록이 포함된 inode/인덱스 블록의 정보를 읽는 과정까지는 데이터 블록 읽기 작업과 동일하다. 본 논문의 개선된 인덱스 구조가 F2FS에 비해서 유리한 것이다. 이렇게 해서 수정할 인덱스 블록의 정보를 읽어 낸 다음에 필요한 부분만 수정해서 다시 기록해야 한다.

파일의 데이터 블록을 쓰고 나면 실제 기록된 PA를 inode/인덱스 블록에 갱신해야 하는데, 이 과정에서 LFS 기반의 파일 시스템은 쓰기 증폭으로 인해서 많은 오버헤드가 발생한다. 끝단(레벨 3)의 인덱스 블록이 수정되면 그 PA를 저장하는 직전 단(레벨 2)의 인덱스 블록을 수정해야 하고, 따라서 그 이전 단(레벨 1)의 인덱스 블록을 수정해야 하고, 마지막에 inode도 수정해야 한다. 인덱스 정보를 수정하기 위해서 총 4회의 쓰기가 필요하다. inode를 수정하기 때문에 NAT 수정은 1회 필요하다. 이에 비해서 F2FS와 본 논문에서 제안한 방식은 데이터 블록의 PA를 기록하고 있는 끝 단의 인덱스 블록(직접 블록인 경우에는 inode)만 수정한다. 인덱스 정보를 수정하는 데에 1회의 쓰기만 필요한 것이다. 인덱스 블록을 1회 수정하므로 NAT의 수정도 1회 필요하다.

Table 2. Write Overhead of a Triple Indirect Blocks

Index Structure	Proposed	F2FS	LFS
Number of NAT Search	2	4	1
Number of inode/Index Block Write	1	1	4
Number of NAT Modification	1	1	1

F2FS는 데이터 블록의 PA가 기록된 끝단의 인덱스 블록만 갱신하므로 쓰기 증폭 문제가 해소된다. 한번의 inode 또는 인덱스 블록의 수정만으로 완료되기 때문이다. 따라서 NAT의 수정도 한 번만 필요하다. 본 논문의 개선된 방식도 L2FS와 동일하게 NAT는 한 번만 수정하면 된다. 그러면서도 개선된 방식은 NAT를 검사하는 회수를 L2FS에 비해서 줄였다.

NAT를 검사하고 수정하는 과정도 면밀하게 분석해 볼 필요가 있다. NAT에는 하나의 파일 시스템에서 허용되는 최대 개수의 inode들을 수용할 수 있어야 하고, 게다가 F2FS에서는 인덱스 블록들 까지 모두 포함해야 한다. 인덱스 블록들의 개수는 간단하게 추정해 볼수 있다. 예를 들어서 파티션의 크기가 4T 바이트이고 블록의 크기가 4K 바이트라면 4T / 4K = 1G 개의 블록들이 있다. 하나의 인덱스 블록에 4K 바이트 크기의 블록 1024개를 기록할 수 있다고 가정하면 1G / 1024 = 1M 개 이상의 인덱스 블록들이 필요하다. NAT에는 여기에다 inode 들을 최대 개수로 수용할 수 있어야 한다. 그러면 수백만 개의 항목들을 NAT에 관리해야 한다.

NAT에는 inode와 인덱스 블록들의 LN으로부터 그들이 기록된 PA를 매핑하는 정보를 관리해야 하는데, 기본적으로 플래시 메모리의 블록들에 기록되어야 한다. 효율적으로 관리하기 위해서는 다양한 형태의 구조를 적용할 수 있겠지만, 수백만 개의 노드들을 수용하려면 최소한 3단계 이상으로 구성해야 한다. 가장 간결하게 저장되어 있다고 해도, 한 블록당 1024개의 PA들을 기록한다고 가정하면 수백만 개의 노드들(Node PA)을 수용하기 위해서는 그림 4와 같이 최소한 3단계가 필요하기 때문이다. NAT 정보는 수시로 수정되기 때문에 실제로는 이보다 더 복잡한 구조로 관리해야 할 것이다.

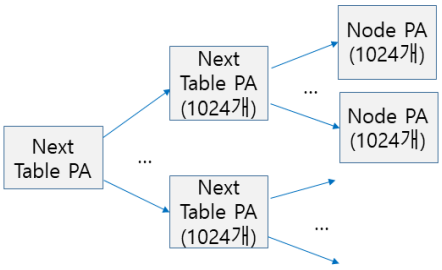

Fig. 4. NAT Mapping Information Structure

제일 상단의 블록은 항상 RAM에 관리한다고 하더라도, NAT를 한번 검사하기 위해서는 2번에 걸쳐서 테이블 블록들을 읽어야 한다. NAT를 수정하는 작업도 2번의 테이블 블록 쓰기 작업이 필요하다. 끝단의 PA를 수정한 다음에 그 상단의 테이블도 수정하고, 다시 RAM에 관리하는 최상단의 테이블을 수정해야 한다. 한번의 NAT 검사에 두 번의 테이블 블록 읽기가 필요하므로, 본 논문의 개선된 인덱스 구조가 NAT를 검사하는 회수를 단축함으로써 실제 플래시 메모리 블록 읽기 회수는 그 두 배의 단축 효과가 있는 것이다.

덧붙여서, NAT에 관리하는 항목 개수도 F2FS에 비해서 줄어든다. F2FS는 모든 인덱스 블록들을 NAT에 관리하지만, 본 논문의 방식은 인덱스 블록들 중에서 끝단의 것만 NAT에 관리하기 때문이다. 따라서 NAT 검사를 위한 시간도 단축될 여지가 있다.

본 논문의 성능 평가는 분석적인 방법으로 진행하였는데, 그 결과로서 실제로 F2FS에 적용하면 성능이 개선될 것은 자명하다. 실제로 적용하였을 때의 개선의 정도는 여러 가지 영향들이 복합적으로 반영될 것이다. 예를 들어서 대부분의 파일 시스템에서는 디스크 캐시(또는 버퍼 캐시)를 적용하므로, 단순히 인덱스 블록 읽기 회수가 반으로 줄었다고 해서 실제로 반으로 줄지는 않을 것이다. 그러나 성능 개선 효과가 있음은 분석적인 방법을 통해서 명백하게 알 수 있다.

IV. Conclusions

플래시 메모리 기반의 저장장치인 SSD는 휴대형 정보기기들뿐만 아니라 대규모 서버 시스템들에도 보편화되고 있다. F2FS는 플래시 메모리의 특성을 반영하여 개발되었으며, 리눅스 운영체제의 표준 파일시스템 중의 하나로 널리 사용되고 있다. F2FS는 성능 향상을 위해서 여러 가지 부분에서 기존의 파일 시스템들을 개선하였다. 그러한 사항들 중의 하나가 파일별 inode 구조에서 데이터 블록들의 정보를 관리하는 인덱스 구조의 개선이다.

본 논문에서는 F2FS의 인덱스 구조를 개선하여 그 성능을 더욱 높이는 방안을 제시하였다. F2FS는 모든 인덱스 블록들을 논리적 번호로 관리하고, 이 번호로부터 실제 기록된 플래시 메모리 블록의 물리적 주소를 매핑하는 테이블을 활용하였다. 그렇게 함으로써 기존의 파일 시스템들에 비해서 쓰기 증폭 효과를 대폭 완화하였다. 본 논문에서는 인덱스 블록들 중에서 끝단의 블록만 논리적 번호를 적용하고, 앞단의 블록들은 물리적 블록 주소를 직접 적용함으로써 성능이 개선됨을 보여주었다. 성능 개선의 핵심은 주소 매핑 테이블을 검색하는 회수를 F2FS에 비해서 줄인 것이다. 또한 매핑 테이블에 관리하는 항목 수도 줄어들기 때문에 부수적인 성능개선효과도 기대할 수 있다. 앞으로 본 논문의 인덱스 구조를 F2FS에 직접 적용하여 구현하고 그 성능을 측정하여 평가하는 작업을 진행할 예정이다.

REFERENCES

[1] A. Gupta, Y. Kim, and B. Urgaonkar. “DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings,” ACM SIGPLAN Notices, Volume 44, Issue 3, pp. 229-240, March 2009. DOI: 10.1145/1508284.1508271

[2] A. B. Bityutskiy, JFFS3 design issues,http://linux-mtd.infradead.org/tech/JFFS3design.pdf, November 27, 2005

[3] C. Lee, et al., “F2FS: a new file system for flash storage,” FAST'15, pp. 273-286, Feb. 2015

[4] M. Rosenblum and J. K. Ousterhout. “The design and implementation of a log-structured file system,” ACM Transactions on Computer Systems (TOCS), Vol. 10, Issue 1, pp.26-52, Feb. 1992. DOI: 10.1145/146941.146943

[5] R. Konishi, et al., “The Linux implementation of a log-structured file system,” ACM SIGOPS Operating Systems Review, Vol. 40, Issue 3, pp. 102-107, July 2006. DOI: 10.1145/1151374.1151375

[6] O. Rodeh, J. Bacik, and C. Mason, “BTRFS: The Linux B-tree Filesystem,” ACM Transactions on Storage (TOS), Vol 9. Issue 3, pp. 1-32, Aug. 2013. DOI: 10.1145/2501620.2501623

[7] M. Larabel, “F2FS File-System Merged Into Linux 3.8 Kernel,” Phoronix, May 25, 2016. https://www.phoronix.com/news/MTI1OTU

[8] A. Artashyan, “ZTE AXON 10 PRO OFFICIALLY UNCOVERED: THE FIRST TO USE F2FS,” Gizchina, May 6, 2019. https://www.gizchina.com/2019/05/06/zte-axon-10-pro-officially-uncovered-the-first-to-use-f2fs

[9] Lihua Yang, et al., “Improving F2FS Performance in Mobile Devices With Adaptive Reserved Space Based on Traceback,” IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 41, no. 1, pp. 169-182, 2022. DOI: 10.1109/TCAD.2021.3054606

[10] Y. Lee, et al., “When F2FS Meets Address Remapping,” ACM HotStorage ‘22, pp. 31-36, June 2022, DOI: 10.1145/3538643.3539755

Authors

Yong-Seok Kim received B.S. degree in Oceanography from Seoul National University, Korea, in 1984, and M.S. and Ph.D. degrees in Electric and Electronics Engineering from KAIST (Korea Advanced Institute of Science and Technology), Korea, in 1986 and 1989, respectively. Dr. Kim is a professor in Department of Computer Engineering at Kangwon National University, Kangwon-do, Korea, from 1995. He was a research staff of KETI (Korea Electronics Technology Institute) in 1994, and KITECH (Korea Institute of Industrial Technology) from 1990 to 1993. He is interested in system software for real-time and embedded systems, and flash memory file systems.