ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 43, No. 6, pp. 627-635, 2016. 6
http://dx.doi.org/10.5626/JOK.2016.43.6.627

로그 구조 파일 시스템의 파일 단편화 해소를 위한 클리닝 기법

(An Efficient Cleaning Scheme for File Defragmentation on Log-Structured File System)

박 종 규 † (Jonggyu Park)　강 동 현 †† (Dong Hyun Kang)　서 의 성 ††† (Euiseong Seo)　엄 영 익 †††† (Young Ik Eom)

요 약 로그 구조 파일 시스템에서는 쓰기 작업을 처리할 때 새로운 블록들이 순차적으로 할당된다. 그러나, 다수의 프로세스가 번갈아가며 동기적 쓰기 작업을 요청할 경우, 파일 시스템 상에서는 각 프로세스가 생성한 파일이 단편화될 수 있다. 이 파일 단편화는 읽기 요청을 처리할 때 다수의 블록 I/O를 발생시키기 때문에 읽기 성능을 저하시킨다. 게다가, 미리 읽기 기능은 한 번에 요청되는 데이터의 양을 증가시킴으로써 성능 저하를 더욱 심화시킨다. 이에, 본 논문에서는 파일 단편화 문제를 해결하기 위해 로그 구조 파일 시스템의 새로운 클리닝 기법을 제안한다. 제안 기법은 로그 구조 파일 시스템의 클리닝 과정 중 유효 데이터 블록을 아이노드 번호 순으로 정렬함으로써 한 파일의 데이터 블록들을 인접하게 재배치한다. 실험 결과, 제안한 클리닝 기법이 클리닝 전에 비해 약 60%의 파일 단편화를 제거하였고, 그 결과로 미리 읽기 기능을 적용했을 때 읽기 성능을 최고 21%까지 향상시키는 것을 확인하였다.

키워드: 로그 구조 파일 시스템, 파일 단편화, 플래시 메모리, 미리 읽기

Abstract When many processes issue write operations alternately on Log-structured File System (LFS), the created files can be fragmented on the file system layer although LFS sequentially allocates new blocks of each process. Unfortunately, this file fragmentation degrades read performance because it increases the number of block I/Os. Additionally, read-ahead operations which increase the number of data to request at a time exacerbates the performance degradation. In this paper, we suggest a new cleaning method on LFS that minimizes file fragmentation. During a cleaning process of LFS, our method sorts valid data blocks by inode numbers before copying the valid blocks to a new segment. This sorting re-locates fragmented blocks contiguously. Our cleaning method experimentally eliminates 60% of file fragmentation as compared to file fragmentation before cleaning. Consequently, our cleaning method improves sequential read throughput by 21% when read-ahead is applied.

Keywords: Log-structured file system, file fragmentation, flash memory, read-ahead

· 이 논문은 2015년도 정부(미래창조과학부)의 재원으로 한국연구재단-차세대정보·컴퓨팅기술개발사업의 지원을 받아 수행된 연구임(No. NRF-2015M3C4A7065696)

† 학생회원 : 성균관대학교 소프트웨어대학
whdrb2722@skku.edu
†† 학생회원 : 성균관대학교 정보통신대학
kkangsu@skku.edu
††† 정 회 원 : 성균관대학교 소프트웨어대학 교수
euiseong@skku.edu
†††† 종신회원 : 성균관대학교 소프트웨어대학 교수
(Sungkyunkwan Univ.)
yieom@skku.edu
(Corresponding author임)

논문접수 : 2016년 3월 8일
(Received 8 March 2016)
논문수정 : 2016년 3월 21일
(Revised 21 March 2016)
심사완료 : 2016년 3월 22일
(Accepted 22 March 2016)

정보과학회논문지 제43권 제6호(2016. 6)

1. 서 론

수년 간, 플래시 메모리는 저전력, 높은 성능, 가벼운 무게 등의 장점으로 인해 스마트폰, 태블릿과 같은 이동식 장치의 저장 장치로 사용되어 왔다. 최근, 플래시 메모리를 기반으로 하는 저장 장치인 Solid State Disk (SSD)의 가격 인하와 성능 최적화는 개인용 컴퓨터, 서버 등의 분야에서도 기존에 널리 사용되어 왔던 하드 디스크(Hard Disk Drive, HDD)를 SSD로 대체시키고 있다. 플래시 메모리는 하드 디스크와는 다른 특성을 가지고 있기 때문에 이를 보완하는 플래시 변환 계층(Flash Translation Layer)이 필요하다. 예를 들어, 플래시 메모리에서는 하드 디스크와는 달리 덮어쓰기가 불가능하기 때문에 데이터가 변경될 때, 기존의 페이지(Page)를 무효화(Invalidation)시키고 프리 페이지에 쓰기 작업을 진행한다[1]. 이와 같은 무효화 작업과 쓰기 작업이 빈번하게 발생할 경우 플래시 메모리의 여유 공간이 부족해진다. 이를 해결하기 위해 플래시 변환 계층에서 가비지 컬렉션(Garbage Collection, GC)을 수행한다. 가비지 컬렉션은 유효 페이지를 새로운 블록으로 이동시키기 위한 추가적인 쓰기 작업을 발생시키기 때문에 플래시 메모리의 성능을 감소시킨다. 특히, 임의 쓰기(Random write)는 가비지 컬렉션 중 발생하는 쓰기 작업의 수를 증가시키므로 플래시 메모리는 임의 쓰기에 취약하다[2].

이러한 플래시 메모리의 취약점을 완화하기 위해 플래시 메모리에 최적화된 파일 시스템들[3-5]이 연구되어 왔다. 그 중 로그 구조 파일 시스템(Log-structured File System, LFS)[6]은 저장장치를 블록으로 구성된 세그먼트 단위로 분할하고 쓰기 요청이 발생하면, 세그먼트 내의 각 블록을 순차적으로 할당한다(append-only 쓰기 정책). 이러한 append-only 쓰기 정책으로 인해 로그 구조 파일 시스템은 대부분의 쓰기 요청을 순차적으로 처리하기 때문에 임의 쓰기에 취약한 플래시 메모리에 적합한 파일 시스템이다. 그러나 이 append-only 쓰기 정책은 파일 단편화(File fragmentation)를 발생시킬 수 있다. 파일 단편화란, 한 파일을 이루는 데이터 블록들의 위치가 인접하지 않고 분산되는 것이다. 예를 들어, 다수의 어플리케이션에서 동시에 동기적(Synchronous) 쓰기 작업을 진행할 경우, 로그 구조 파일 시스템에서는 요청된 쓰기 작업들을 어플리케이션의 구분 없이 세그먼트에 순차적으로 블록을 할당한다. 그렇기 때문에 각 어플리케이션에서 생성한 파일은 그 데이터 블록들의 위치가 분산될 수 있다.

기존의 많은 연구들은 플래시 메모리에서의 파일 단편화 문제를 고려하지 않았다. 왜냐하면, 플래시 메모리는 하드 디스크와 다르게 탐색시간이 없고, 파일 시스템 상의 데이터 위치와 플래시 메모리 상의 데이터 위치가 다르기 때문이다. 그러나 플래시 메모리에서도 파일 단편화는 커널(Kernel)에 오버헤드(Overhead)를 발생함으로써 읽기 성능을 감소시킨다. 어플리케이션이 파일을 읽을 때 커널은 읽기 요청을 처리하기 위해 bio라는 구조체를 생성한다[7]. bio란 블록 I/O를 처리하는 기본적인 단위로서, 파일 시스템의 논리 주소(Logical address) 상에서 인접한 블록을 표현해주는 구조체이다. 하나의 bio는 파일 시스템 상에서 하나의 인접한 블록 그룹만 표현할 수 있기 때문에 요청된 블록들이 인접하지 않으면 떨어져있는 블록마다 bio를 생성해야한다. 그래서 파일 단편화는 생성해야 할 bio의 개수를 증가시키기 때문에 읽기 성능을 감소시킨다. 게다가 현재 리눅스 커널에서 제공하는 미리 읽기 기능을 적용하면 파일 단편화 문제는 더욱 심각해진다. 미리 읽기란, 어플리케이션이 파일 내의 데이터에 접근할 때 커널이 공간 구역성(Spatial locality)을 활용하여 그 다음 파일 오프셋을 가지는 데이터를 함께 메모리로 로드(Load)하는 것이다. 이때, 미리 읽기는 요청하는 데이터의 양을 증가시키므로 파일 단편화가 발생했을 경우, 커널이 훨씬 더 많은 bio를 생성해야 한다. 결과적으로 이는 미리 읽기로 인한 성능 향상 효과를 저하시킨다.

파일 단편화에 관한 대부분의 연구들[8-10]은 하드 디스크를 기반으로 이루어져 있다. 그 중 단편화 제거 기법(File defragmentation, 조각모음)은 이미 분산되어 있는 파일 블록들을 인접하게 재배치시키는 기법이다. 그러나 이 기법은 블록을 재배치하는 과정 중 추가적인 블록 I/O를 발생시키므로 재기록 가능 횟수가 정해져있는 플래시 메모리의 수명을 감소시킨다. 이 문제를 해결하기 위해 본 논문에서는 로그 구조 파일 시스템을 수정하였다. 로그 구조 파일 시스템은 기존의 블록에 덮어쓰기를 허용하지 않기 때문에 프리 세그먼트 확보를 위한 세그먼트 단위의 클리닝 작업을 수행한다[11,12]. 세그먼트 단위의 클리닝 작업은 희생 세그먼트 내의 유효 블록들을 새로운 세그먼트에 복사하고, 그 희생 세그먼트를 프리 세그먼트로 지정하는 것이다. 기존의 클리닝 작업은 유효 데이터 블록을 읽은 순서 그대로 새로운 세그먼트에 복사를 하므로, 파일 단편화가 그대로 상속되었다. 그래서 본 논문에서는 로그 구조 파일 시스템의 클리닝 과정을 수정하여 파일 기반 클리닝을 제안한다. 제안한 기법은 기존의 방식과는 다르게 유효 데이터 블록을 아이노드(inode) 번호 순으로 정렬한 후 새로운 세그먼트에 복사한다. 각 파일은 고유의 아이노드 번호를 소유하고 있기 때문에, 아이노드 번호를 활용한 정렬은 한 세그먼트 내의 데이터 블록을 같은 파일의 다른 데

이터 블록들과 인접하게 위치시킨다. 이 기법은 기존 클리닝과 비교하여 추가적인 블록 I/O를 발생시키지 않기 때문에, 플래시 메모리의 수명을 감소시키지 않는다.

본 논문은 다음과 같이 구성되어있다. 2절에서는 파일 단편화, 로그 구조 파일시스템과 같은 본 논문의 주제와 관련된 배경지식을 살펴본다. 3절에서는 우리가 제안한 파일 기반의 클리닝 기법의 원리 및 과정을 설명한다. 4절에서는 실험을 통해 로그 구조 파일 시스템의 파일 단편화 문제점을 증명하고, 제안한 기법의 효용성을 검증한다. 마지막으로 5절에서 결론을 맺는다.

2. 관련 연구

본 절에서는 파일 시스템 레이아웃, 쓰기 정책, 클리닝 등의 로그 구조 파일 시스템의 기본적인 원리와 로그 구조 파일 시스템에서 파일 단편화의 발생 원인과 사례를 살펴본다. 그리고 커널의 블록 I/O 과정을 분석하여, 파일 단편화가 읽기 성능에 미치는 영향을 미리 읽기 기능과 연관시켜 분석한다.

2.1 로그 구조 파일 시스템

먼저, 로그 구조 파일 시스템의 레이아웃을 살펴보자. 기존의 EXT4와 같은 파일 시스템은 블록의 종류에 따라 영역을 구분하였다. 예를 들어, 아이노드 블록, 데이터 블록, 슈퍼 블록 등과 같은 블록의 종류에 따라 각각 따로 영역을 지정한다.

반면에, 로그 구조 파일 시스템은 그림 1과 같이 블록의 종류에 상관없이 디스크를 세그먼트 단위로 분할한다. 분할된 세그먼트는 여러 개의 블록으로 이루어져 있으며, 블록의 종류는 파일의 정보를 나타내는 아이노드 블록(Inode block), 파일의 데이터를 나타내는 데이터 블록(Data block), 데이터 블록의 위치를 나타내는 간접 블록(Indirect block) 등이 있다.

일반적인 로그 구조 파일 시스템은 append-only 쓰기 정책을 사용한다. 그래서 로그 구조 파일 시스템에서는 쓰기 작업이 요청되면, 마지막으로 할당했던 블록의 바로 다음 위치에 새로운 블록을 할당한다. 본 논문에서는 이 위치를 LFS 쓰기 지점이라 부른다. 이 독특한 쓰기 정책으로 인해 로그 구조 파일 시스템은 대부분의 쓰기 요청을 순차적으로 처리한다.

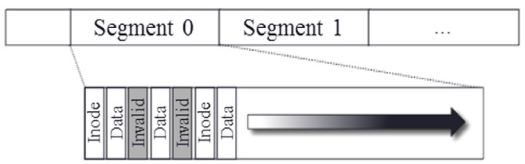

그림 1 로그 구조 파일 시스템의 layout
Fig. 1 Log-structured File System layout

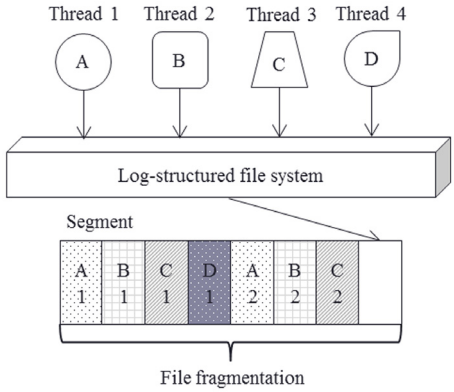

그림 2 로그 구조 파일 시스템에서 여러 개의 동기적 쓰기로 인한 파일 단편화
Fig. 2 File fragmentation caused by concurrent synchronous writes in LFS

그러나 이 append-only 쓰기 정책은 새로운 블록을 할당하는 위치가 LFS 쓰기 지점으로 정해져 있기 때문에 파일 단편화를 야기할 수 있다. 예를 들어, 한 파일에서 일부분의 데이터가 수정되는 경우, 로그 구조 파일 시스템은 기존의 데이터 블록을 무효화시키고, 새로운 데이터 블록을 LFS 쓰기 지점에 할당한다. 그러면 수정된 데이터 블록은 수정되지 않은 데이터 블록들과 떨어지게 되므로 파일 단편화가 발생한다. 이 뿐만 아니라, 여러 스레드가 동기적 쓰기를 요청할 경우에도 파일 단편화가 발생한다. 그림 2에서 각 블록의 영문자는 파일 이름, 숫자는 파일 오프셋을 나타낸다. 그림 2와 같이, 4개의 스레드에서 동기적 쓰기를 요청할 경우, 여러 스레드가 쓰기 작업을 번갈아 진행한다. 이때, 로그 구조 파일 시스템에서는 각 파일이 번갈아 가면서 LFS 쓰기 지점에 블록을 할당받기 때문에 각 파일의 데이터 블록은 인접하게 위치하지 않는다.

로그 구조 파일 시스템은 또 다른 특징은 클리닝 작업이다. 로그 구조 파일 시스템은 덮어쓰기를 허용하지 않기 때문에 파일이 수정됐을 때, 기존의 블록을 무효화하고 새로운 블록을 할당한다. 그래서 일정 수준의 프리 세그먼트를 확보하기 위해 무효 블록을 수집하는 클리닝 작업이 필요하다. 클리닝 과정에는 포어그라운드 클리닝과 백그라운드 클리닝이 있다. 우리는 본 논문에서 포어그라운드 클리닝만 다룬다. 로그 구조 파일 시스템의 세그먼트 클리닝 과정은 클리닝 스레드가 희생 세그먼트의 유효 블록들을 새로운 세그먼트에 복사하고, 희생 세그먼트를 프리 세그먼트로 변경시켜준다. 클리닝 스레드가 수행하는 자세한 과정은 다음과 같다.

1. 클리닝 정책에 따라 희생 세그먼트를 지정한다.

2. 희생 세그먼트 내의 블록들을 순차적으로 메인 메모리에 로드한다.
3. 로드한 블록이 유효 블록인지 확인한다.
4. 그 블록이 유효 블록이면 새로운 세그먼트에 복사한다.
5. 희생 세그먼트의 모든 유효 블록이 복사되면, 프리 세그먼트로 지정한다.
6. 위의 과정을 설정된 프리 세그먼트 개수만큼 확보될 때까지 반복한다.

클리닝 정책에 따라 클리닝 스레드가 희생 세그먼트를 지정할 때, 클리닝 오버헤드를 줄이기 위해 여러 가지 정책을 사용한다. 일반적으로 greedy 정책과 cost-benefit 정책이 있다. greedy 정책은 복사해야 할 유효 블록이 가장 적은 세그먼트를 선택하여 클리닝 오버헤드를 줄이는 정책이다. 그리고 cost-benefit 정책은 블록의 참조성을 고려하여 수명이 같은 블록끼리 그룹화 시켜주는 기법이다. 이렇게 클리닝 정책에 따라 희생 세그먼트가 선택이 되면, 클리닝 스레드는 유효 블록들을 순차적으로 새로운 세그먼트에 복사한다. 희생 세그먼트의 모든 유효 블록이 복사되면 클리닝 스레드는 희생 세그먼트를 프리 세그먼트로 지정하고 일정한 수준의 프리 세그먼트가 확보될 때까지 위 과정을 반복한다. 이러한 클리닝 과정을 통해서 로그 구조 파일 시스템에서 일정 수의 프리 세그먼트를 유지하고 있다.

2.2 파일 단편화

파일 단편화란, 그림 3과 같이 파일을 이루는 데이터 블록들이 파일 시스템 상에서 인접하지 않고, 분산되어 위치하는 것이다. 하드 디스크에서는 데이터에 접근할 때 탐색 시간이 필요하기 때문에 파일 단편화가 발생하면 탐색 시간 증가로 인해 블록 I/O의 성능이 감소한다. 하지만 플래시 메모리에서는 하드 디스크에 존재하는 탐색 시간이 필요하지 않고, 파일 시스템 상의 데이터 위치와 플래시 메모리 상의 데이터 위치가 다르기 때문에 많은 연구들이 플래시 메모리에서의 파일 단편화 문제를 고려하지 않는다. 그래서 대다수의 파일 단편화에 관련된 연구들은 하드 디스크에 최적화되어 있다.

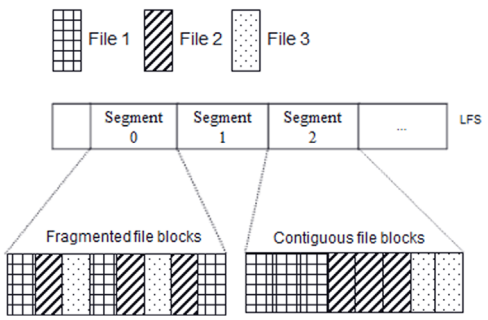

그림 3 파일 단편화와 비-단편화

Fig. 3 File fragmentation and non-fragmentation

예를 들어, 파일 단편화 제거 기법은 분산된 데이터 블록을 재배치시켜서 인접하게 만드는 것이다. 재배치 과정 중 추가적인 쓰기 작업이 발생하기 때문에 재기록 가능 횟수가 한정된 플래시 메모리의 수명을 떨어뜨린다.

2.3 읽기 작업과 bio

현재 리눅스 커널에서는 블록 읽기, 블록 쓰기와 같은 블록 I/O를 표현할 때 bio 구조체를 사용한다. bio 구조체란 어플리케이션이 블록 I/O를 요청할 때, 커널에서 생성하는 자료구조로써 파일 시스템의 논리 주소 상에서 인접한 블록 그룹을 표현한다.

그러나 그림 4와 같이, 어플리케이션에서 인접한 오프셋의 파일 데이터를 요청하더라도 실제 논리 주소가 인접하지 않으면, 분산된 데이터 블록마다 따로 bio를 생성해야 한다. 그렇기 때문에 어플리케이션이 파일 데이터를 요청할 때, 파일 단편화는 생성해야 할 bio의 개수를 증가시킴으로써 읽기 성능을 저하시킨다. 이러한 파일 단편화의 성능 저하는 미리 읽기가 적용되면 더욱 심각해진다.

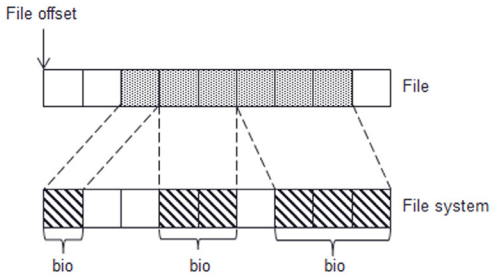

그림 4 파일 오프셋과 파일 데이터 블록의 논리적 주소와의 관계

Fig. 4 Relationship between file offset in a file and block number in file system

2.4 미리 읽기

미리 읽기란, 메인 메모리에 비해 상대적으로 느린 디스크의 성능을 보완하기 위해서 커널에서 제공하는 기능이다. 어플리케이션이 한 파일의 데이터를 로드할 때, 다음 파일 오프셋을 가진 데이터를 함께 메인 메모리로 로드한다. 그 후, 어플리케이션이 미리 로드된 데이터들을 요청할 때, 추가적인 블록 I/O 없이 메모리 상에서 그 요청이 처리되므로 읽기 성능이 향상된다. 특히, 어플리케이션이 순차적으로 데이터를 요청한다면, 대부분의 읽기 요청이 메모리 상에서 처리가 되므로 높은 성능 향상을 얻을 수 있다.

이러한 원리로 미리 읽기는 한 개의 읽기 요청이 포함하는 데이터의 양을 증가시킨다. 만약, 요청된 블록들이

파일 시스템 상에서 인접하지 않다면, 미리 읽기로 인해 늘어난 데이터의 양만큼 커널은 추가적으로 bio를 생성해야 하므로 파일 단편화로 인한 성능 감소는 증가한다.

3. 파일 기반 클리닝

2절에서 언급하였듯이, 로그 구조 파일 시스템의 append-only 쓰기 방식은 파일 단편화를 야기한다. 그리고 파일 단편화는 하드 디스크뿐만 아니라 플래시 메모리에서도 읽기 성능을 저하시킨다. 특히 미리 읽기가 적용됐을 경우 그 성능 저하는 심화된다. 그래서 본 논문에서 로그 구조 파일시스템에서 플래시 메모리의 수명 저하 없이 파일 단편화를 해소하는 파일 기반 클리닝 기법을 제시한다. 파일 기반 클리닝 기법은 플래시 메모리의 수명을 고려하여 추가적인 블록 I/O 발생을 방지하기 위해 기존의 클리닝 기법을 활용한다.

기존의 클리닝 기법은 유효 블록을 새로운 세그먼트에 순차적으로 복사한다. 이렇게 유효 블록을 순차적인 복사하면 그림 5와 같이 파일 단편화가 존재할 경우 그 파일 단편화가 새로운 세그먼트에도 그대로 상속이 되기 때문에 파일 단편화 문제를 해결할 수 없다. 한 파일은 각각 한 개의 고유한 아이노드 번호를 소유한다. 우리는 이 원리를 활용하여 파일 기반 클리닝 기법을 제시한다. 파일 기반 클리닝 기법이란, 클리닝 과정 중에서 클리닝 스레드가 유효 데이터 블록을 새로운 세그먼트로 바로 이동시키지 않고, 아이노드 번호 순으로 정렬한 후에 이동시켜서 파일 단편화를 해결하는 기법이다.

우리는 이 기법에서 정렬 작업을 하기 전까지 유효 데이터 블록을 수집하기 위해 유효 블록 큐(Valid Block Queue, VBQueue)를 제안한다. 자세한 과정은 다음과 같다.

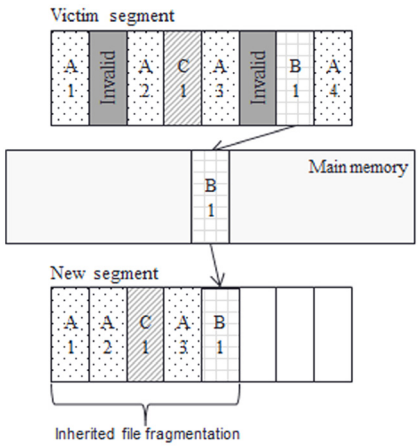

그림 5 기존의 클리닝에서는 상속되는 파일 단편화
Fig. 5 Inherited file fragmentation in existing cleaning process

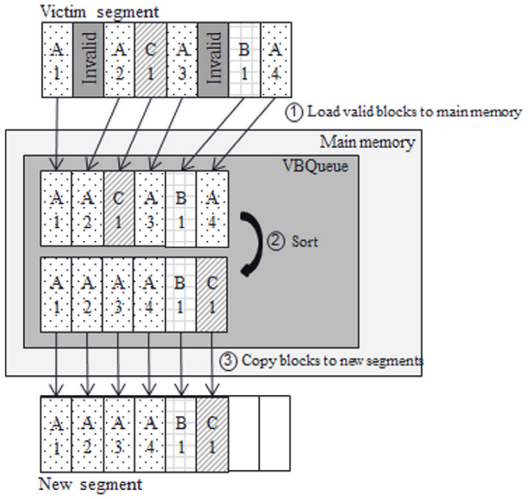

그림 6 파일 기반 클리닝 기법
Fig. 6 File-Aware Cleaning method

먼저, 기존의 클리닝 방식과 동일하게 클리닝 정책에 따라 클리닝 스레드가 희생 세그먼트를 선정한다. 그 후, 클리닝 스레드가 희생 세그먼트에 존재하는 블록들의 유효성을 확인하고 유효 데이터 블록들을 메인 메모리의 VBQueue로 복사한다. 희생 세그먼트의 모든 유효 데이터 블록들이 VBQueue에 복사되면, 클리닝 스레드는 VBQueue에 있는 데이터 블록들을 각 블록이 가리키는 파일의 아이노드 번호 순으로 정렬을 한다. 아이노드 번호는 파일마다 고유하므로 같은 파일의 데이터 블록은 인접하게 위치하게 된다. 그리고 이 과정은 추가적인 블록 I/O 없이 메모리 상에서만 처리되기 때문에 클리닝 작업에 큰 오버헤드를 주지 않는다. VBQueue의 모든 블록이 정렬되면 새로운 세그먼트에 모두 이동시킨다. 그 후 과정은 기존 클리닝과 동일하다. 우리는 리눅스 커널에서 제공하는 list 자료구조를 활용하여 VBQueue를 구현하였으며, 정렬 작업은 커널에서 제공하는 list_sort 함수를 사용하였다. list_sort 함수는 정렬 과정에서 직접적인 데이터의 이동이 아닌 인덱스만을 변경시키기 때문에 정렬 과정의 오버헤드를 감소시킨다. 결과적으로 파일 기반 클리닝 기법은 그림 6과 같이 로그 구조 파일 시스템에서 플래시 메모리의 수명 감소 없이 파일 단편화 문제를 해결한다.

4. 실 험

4.1 실험 환경

본 논문에서 제안한 파일 기반 클리닝의 성능을 측정하기 위해 리눅스 커널 4.4 버전의 F2FS[13]에서 파일 기반 클리닝을 구현하였다. F2FS는 리눅스 커널에서 지

원하는 로그 구조 파일 시스템 중 하나로서, 아이노드 블록과 데이터 블록을 구분해서 각각 다른 세그먼트에 할당하는 특성이 있다. 그래서 우리는 데이터 블록 세그먼트의 경우에만 제안한 클리닝 기법을 적용하였다. 우리는 이번 실험에서 16GB SD card를 플래시 메모리 저장 장치로 사용하였다. 그리고 실험을 위한 벤치마크로는 현재 파일 시스템 벤치마크로 많이 쓰이는 IOzone[14]을 이용하였다. 우리는 8개의 스레드가 동시에 동기적으로 각각 한 개의 파일을 생성하는 실험을 하였다. 하나의 파일의 크기는 1.6GB이고, record 크기는 32KB로 설정하였다. 로그 구조 파일 시스템의 append-only 쓰기 정책으로 인한 파일 단편화 문제를 확인하기 위해 append-only 쓰기 정책을 사용하지 않는 EXT4에도 동일한 실험을 진행하여 비교했다. 그리고 순차적 읽기(Sequential read) 성능을 측정하여 파일 단편화가 읽기 성능에 끼치는 영향을 분석하였다. 또한, 우리가 제시한 기법의 효용성을 검증하기 위해 클리닝 작업을 발생시켜서 기존의 클리닝 방식과 파일 단편화 해소 정도와 읽기 성능을 측정하였다. 마지막으로 파일 기반 클리닝이 기존의 클리닝과 비교하여 어느 정도의 오버헤드가 발생하는지 분석해보았다.

4.2 실험 결과

4.2.1 로그 구조 파일 시스템과 파일 단편화

그림 7은 EXT4와 F2FS에서 동일하게 IOzone으로 실험을 하였을 때, 파일 단편화의 정도를 나타낸 것이다. X축은 파일 시스템 종류이고, Y 축은 8개의 파일 중 한 개의 파일을 분석하여 파일 시스템 상에서 그 파일을 이루는 데이터 블록 그룹(Data block group)의 수를 나타낸 것이다. 데이터 블록 그룹이란, 하나의 인접한 데이터 블록의 모음이다. 예를 들어, 그림 8과 같은 상황일 때 파일 1의 경우 블록이 8개로 이루어져 있지만 블록 4개는 인접하게 위치한다. 그래서 데이터 블록 그룹의 개수는 8개가 아니라 5개가 된다. 우리는 이 값을 파일 단편화 정도를 나타내는 단위로 사용한다.

IOzone으로 8개의 파일을 동기적으로 생성한 결과, EXT4의 경우 1.6GB의 파일이 파일 시스템 상에서 208개의 데이터 블록 그룹으로 분리되지만, F2FS의 경우 1.6GB 파일이 51200개의 데이터 블록 그룹으로 분리된다. 그 이유는 로그 구조 파일 시스템은 append-only 쓰기 정책을 사용하기 때문에 새로운 블록이 할당되는 위치(LFS 쓰기 지점)가 정해져 있다. 그래서 8개의 스레드가 동시에 파일을 쓸 경우, 8개의 파일이 번갈아 가면서 LFS 쓰기 지점의 블록을 할당 받기 때문에 각 파일은 인접한 데이터 블록 그룹을 생성할 수 없다.

4.2.2 파일 단편화와 읽기 성능

그림 7과 같이 F2FS에서는 독특한 쓰기 정책으로 인

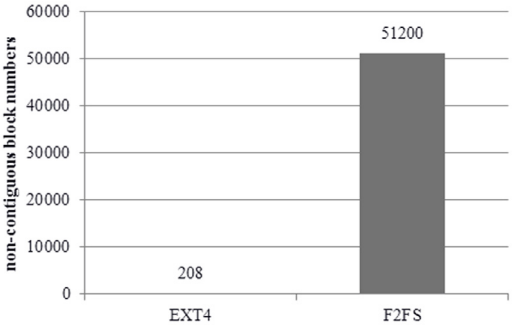

그림 7 EXT4와 F2FS의 파일 단편화 비교

Fig. 7 File framentation degree of EXT4 vs. F2FS

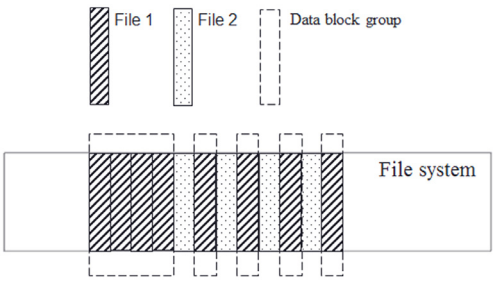

그림 8 데이터 블록 그룹

Fig. 8 Data block group

해 파일 단편화가 발생한다. 우리는 파일 단편화가 미리 읽기로 인한 성능 향상에 미치는 영향을 분석하기 위해 한 파일을 처음부터 끝까지 순차적으로 읽고 그 성능을 측정해 보았다. 실험은 각 파일 시스템마다 미리 읽기를 적용하지 않았을 때와, 적용했을 때 두 가지 경우를 측정하였고 record 크기는 32KB이다. 미리 읽기의 크기는 기본 값인 128KB로 설정하였다. 그림 9에서 X축은 파일 시스템의 종류와 미리 읽기의 크기를 나타낸다. Y축은 단편화된 파일 전체를 순차적 읽기를 했을 때 시간당 처리량(KB/sec)을 나타낸다. IOzone에서 파일을 생성할 때 32KB의 record 크기로 쓰기 작업을 요청하기 때문에 F2FS에서는 32KB 단위로 단편화된 데이터 블록 그룹이 생성된다. 이때, 미리 읽기를 적용하지 않은 경우에는 한 개의 읽기 요청의 크기가 32KB로 동일하기 때문에 파일 단편화의 영향을 받지 않는다. 그래서 EXT4와 F2FS에서 생성하는 bio 개수는 차이가 없으므로 EXT4와 F2FS의 순차적 읽기 성능은 비슷하다. 그러나 미리 읽기를 적용했을 때는 128KB만큼 더 큰 단위로 순차적 읽기가 요청되기 때문에 파일 단편화의 영향이 커지므로 F2FS에서는 EXT4보다 더 많은 bio가 생성된다. 결과적으로 F2FS에서는 EXT4와 비교하여 약 32% 낮은 순차적 읽기 성능을 보인다.

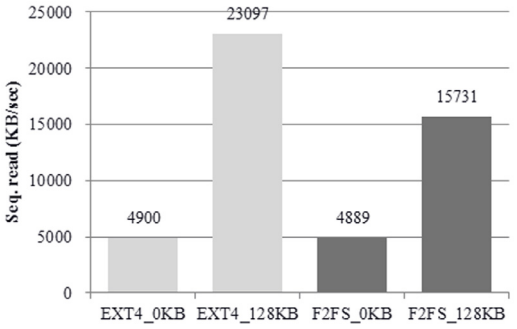

그림 9 EXT4와 F2FS의 순차적 읽기 성능 차이
Fig. 9 Sequential read performance of EXT4 vs. F2FS

4.2.3 파일 기반 클리닝의 순차적 읽기 성능 향상 효과

파일 기반 클리닝의 성능 향상 효과를 측정하기 위해 8개의 파일 중 1개의 파일을 삭제하고 클리닝을 발생시켰다. 파일을 삭제한 이유는 실험 파일들을 포함하는 세그먼트에 무효화한 블록을 생성하여 그 세그먼트가 희생 세그먼트로 선정되기 위함이다. 그리고 클리닝 발생 후 실험 4.2.2와 동일하게 미리 읽기 기능을 적용하고 한 개의 파일을 순차적으로 읽고 성능을 측정해보았다. 그림 10의 X축은 각각 클리닝 기법을 나타낸다. 왼쪽은 기존의 클리닝 기법이고 오른쪽은 제안한 클리닝 기법이다. Y축은 순차적 읽기의 시간 당 처리량(KB/sec)을 나타낸다.

기존의 클리닝 기법은 파일 삭제로 인하여 발생한 무효화한 블록을 제거함으로써 클리닝 전과 대비하여 9% 정도의 읽기 성능향상이 있었다. 그러나 기존의 클리닝은 파일 단편화를 충분히 해소하지 못하므로 성능 향상이 미비했다. 우리가 제안한 파일 기반 클리닝은 클리닝 과정 동안 분산되어있던 데이터 블록들이 인접하게 위치하도록 재배치시킨다. 그 결과, 우리가 제안한 클리닝 기법은 클리닝 전보다 약 32%의 순차적 읽기 성능이 향상시켰다. 기존의 클리닝 기법보다는 약 21% 정도 높은 순차적 읽기 성능을 보였다.

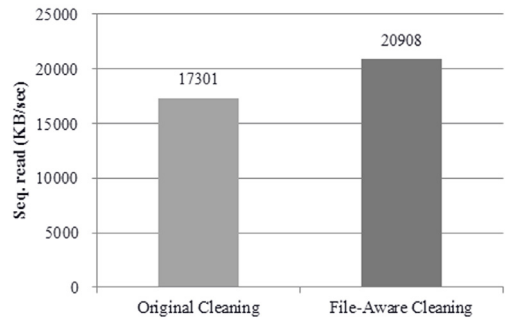

그림 10 기존의 클리닝 기법과 파일 기반 클리닝 기법의 순차적 읽기 성능 비교
Fig. 10 Sequential read performance of Original Cleaning vs. File-Aware Cleaning

4.2.4 파일 기반 클리닝과 파일 단편화 제거 효과

파일 기반 클리닝을 통한 순차적 성능 향상의 원인이 파일 단편화의 해소임을 확인하기 위해 4.2.1과 같이 데이터 블록 그룹의 개수를 확인하였다.

그림 11에서 X축은 클리닝 종류를 나타내는 것이고, Y축은 한 파일의 데이터 블록 그룹의 개수를 나타낸 것이다. 기존의 클리닝 기법으로는 클리닝 전보다 데이터 블록 그룹의 개수가 약 5% 감소한 반면, 우리가 제시한 기법은 클리닝 전보다 데이터 블록 그룹 개수를 약 60% 감소시켰다.

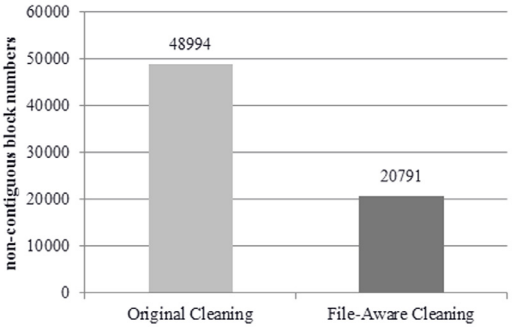

그림 11 기존의 클리닝 기법과 파일 기반 클리닝 기법의 파일 단편화 제거 정도 비교
Fig. 11 File fragmentation degree of Original Cleaning vs. File-Aware Cleaning

4.2.5 파일 기반 클리닝의 클리닝 오버헤드

파일 기반 클리닝은 클리닝 과정 중에 정렬 작업을 수행하기 때문에 그 과정에서 추가적인 오버헤드가 발생할 수 있다. 그래서 우리는 기존의 클리닝과 파일 기반 클리닝의 평균 클리닝 수행 시간을 비교해보았다. 평균 클리닝 수행 시간은 클리닝 작업이 발생할 때 클리닝 작업의 시작시간과 종료시간의 차이를 이용하여 측정하였다. 그림 12는 클리닝 오버헤드 측정 실험에서 발생하는 각 클리닝 기법의 평균 수행 시간이다. 측정 결과, 제안한 클리닝 기법의 수행 시간이 정렬 작업의 오버헤드로 인해 기존의 클리닝 기법에 비해 평균적으로 약 154μs 증가하였다.

5. 결 론

로그 구조 파일 시스템은 append-only 쓰기 정책을 사용하기 때문에 쓰기 작업을 수행할 때 순차적으로 새로운 블록을 할당한다. 그러나, 다수의 프로세스가 동기

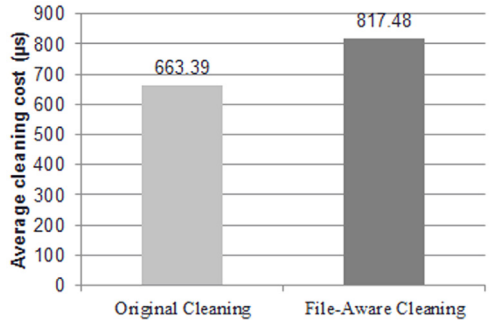

그림 12 기존의 클리닝 기법과 파일 기반 클리닝 기법의 클리닝 오버헤드 비교

Fig. 12 Cleaning overhead of Original Cleaning vs. File-Aware Cleaning

적으로 번갈아가며 쓰기 작업을 요청할 경우에는 파일 시스템 상에서는 생성된 파일들이 단편화될 수 있다. 이 파일 단편화는 읽기 요청을 처리하는 과정 중에 다수의 블록 I/O를 발생시킬 수 있기 때문에 읽기 성능에 악영향을 미친다. 게다가 미리 읽기는 한 번에 요청하는 데이터의 양을 증가시킴으로써 읽기 성능 저하를 심화시킨다. 이에, 본 논문에서는 로그 구조 파일 시스템의 클리닝 방식을 차용하여 이 문제를 해결하였다. 본 논문에서 제안한 파일 기반 클리닝 기법은 클리닝 과정에서 VBQueue를 이용하여 유효 데이터 블록을 아이노드 번호 순으로 정렬함으로써 세그먼트 내의 파일 단편화를 최소화하였다. 실험 결과, 파일 기반 클리닝을 통해 파일 단편화가 약 60% 감소하였으며, 미리 읽기 적용 시 순차적 읽기 성능을 기존 클리닝 대비 약 21% 향상시켰다.

본 논문에서는 파일 단편화와 읽기 성능만을 고려하였다. 하지만 읽기 성능은 파일의 할당 크기, 세그먼트의 크기, 플래시 블록 및 페이지와의 상관관계 등에 의해 차이가 날 수 있기 때문에 이러한 요소들을 고려하여 추후 연구를 진행할 예정이다. 그리고 실험 또한 본 논문에서는 IOzone 만을 사용하였지만 추후 연구에서는 실제 시스템 환경을 반영한 다양한 실험을 진행하여 연구의 타당성을 높일 예정이다.

References

[1] Y. Kang, K. Han, and D. Shin, "Improving the Performance of Log-Structured File Systems Using Logical Copy Technique," *Proc. of the KIISE Korea Computer Congress*, pp. 1258-1260, 2008. (in Korean)

[2] C. Min, K. Kim, H. Cho, S. Lee, and Y. Eom, "SFS: Random Write Considered Harmful in Solid State Drives," *Proc. of USENIX Conference on File and Storage Technologies*, pp. 12, 2012.

[3] J. Engel and R. Mertens, "LogFS-finally a Scalable Flash File System," *Proc. of International Linux System Technology Conference*, pp. 135-142, 2005.

[4] A. Schierl, G. Schellhom, D. Haneberg, and W. Reif, "Abstract Specification of the UBIFS File System for Flash Memory," *Proc. of International Symposium on Formal Methods*, pp. 190-206, 2009.

[5] S. Lim and K. Park, "An Efficient NAND Flash File System for Flash Memory Storage," *IEEE Transactions on Computers*, Vol. 55, No. 7, pp. 906-912, 2006.

[6] 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.

[7] L. Robert, *Linux Kernel Development*, 3rd edition, Pearson Education, 2010.

[8] V. Steve, C. Frost, and E. Kohler, "Reducing Seek Overhead with Application-Directed Prefetching," *Proc. of USENIX Annual Technical Conference*, pp. 24, 2009.

[9] M. Kaczmarczyk, M. Barczynski, W. Kilian, and C. Dubnicki, "Reducing Impact of Data Fragmentation Caused by In-line Deduplication," *Proc. of ACM International Systems and Storage Conference*, pp. 15, 2012.

[10] K. A. Smith and M. I. Seltzer, "File System Aging-Increasing the Relevance of File System Benchmarks," *ACM SIGMETRICS Performance Evaluation Review*, Vol. 25, No. 1, pp. 203-213, 1997.

[11] R. Mendel and J. K. Ousterhout, "The LFS Storage Manager," *Proc. of USENIX Summer 1990 Technical Conference*, pp. 315-324, 1990.

[12] T. Blackwell, J. Harris, and M. I. Seltzer, "Heuristic Cleaning Algorithms in Log-Structured File Systems," *Proc. of USENIX 1995 Technical Conference*, pp. 277-288, 1995.

[13] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," *Proc. of USENIX Conference on File and Storage Technologies*, pp. 273-286, 2015.

[14] W. D. Norcott and D. Cappas. (2016, Jan 23). IOzone filesystem benchmark [Online]. Available: http://www.iozone.org/ (downloaded 2016, Feb. 12)

박 종 규

2014년 성균관대학교 소프트웨어학과(학사). 2014년~현재 성균관대학교 소프트웨어플랫폼학과 석사과정. 관심분야는 스토리지 시스템, 운영체제

강 동 현
정보과학회논문지
제 43 권 제 1 호 참조

서 의 성
2000년 KAIST 전산학과(학사). 2002년 KAIST 전산학과(석사). 2007년 KAIST 전산학과(박사). 2007년~2009년 Penn State Univ., Dept. of CSE(Research Associate). 2009년~2012년 UNIST 전기전자컴퓨터공학부(조교수). 2012년~현재 성균관대학교 소프트웨어대학(부교수). 관심분야는 저전력 컴퓨팅, 클라우드 컴퓨팅, 시스템 소프트웨어

엄 영 익
정보과학회논문지
제 43 권 제 1 호 참조