Journal of Digital Contents Society Vol. 12 No. 2 Jun. 2011(pp. 241-251)

SSD에서의 시맨틱 파일 검색을 위한 확장된 속성 제공의 로그기반 파일시스템

기안호*, 강수용**

요 약

운영체제의 다른 부분이나 저장장치, 매체의 변화에 비해 파일시스템은 지난 수십 년 간 그 발전 속도가 더딘 편이다. 그러나 데이터의 증가에 따라 파일의 개수는 기하급수적으로 증가하고 있으며, 이렇게 늘어난 파일들에 대해 검색을 효율적으로 수행하기 위한 새로운 파일시스템 구조에 대한 연구가 최근 파일에 대한 시맨틱 검색을 하고자 하는 요구와 맞물려 주목 받고 있다. 하지만 이러한 연구는 저장장치와 바로 맞닿는 계층이 아닌 더 상위 계층에서만 이루어지고 있어 기존의 하드디스크와는 다른 특성을 지니는 플래시 메모리 기반의 저장장치인 SSD에 최적화를 시키기 위한 노력으로 이어지진 않았다. 논문에서는 다중 로깅 지점이라는 SSD의 특성을 활용한 로그기반 파일시스템이 SSD에서 얻는 성능상의 이점을 활용하여 새로운 요구사항인 시맨틱 파일 검색까지 추가 비용 없이 지원하는 파일시스템을 제안한다.

Attribute-Rich Log-Structured Filesystem for Semantic File Search on SSD

Anho Ki*, Sooyong Kang**

Abstract

During the last decades, other parts of operating systems, storage devices, and media are changed steadily, whereas filesystem is changed little. As data is grown bigger, the number of files to be managed also increases in geometrically. Researches about new filesystem schemes are being done widely to support these files efficiently. In web document search area, there are many researches about finding meaningful documents using semantic search. Many researches tried to apply these schemes, which is been proven in web document search previously, to filesystems. But they've focused only on higher layer of filesystem, that is not related seriously to storage media. Therefore they're not well tuned to physical characteristics of new flash memory based SSD which has different features against traditional HDD. We enhance log structured filesystem, that is already well known to work better in SSD, by putting semantic search scheme to and with multi logging point.

Keywords : SSD, Semantic Filesystem, Log-Structured Filesystem

1. 서론

파일시스템의 인터페이스는 운영체제의 다른 부분과는 달리 지난 수십 년간 거의 변화하지 않고 있다. 기존의 디렉터리 구조와 약간의 파일에 대한 속성을 담고 있는 메타데이터 구조는 현재의 인터페이스로도 충분히 다룰 수 있었다.

그러나 그와 달리 저장장치에 대한 주변 여건은 계속해서 변화하고 있다. 대용량 멀티미디어 자료가 계속 늘어가고 있으며 그에 따라 저장장치의 용량도 계속 커지고 있다[1]. 또한 새로운 매체인 플래시 메모리의 등장으로 하드디스크와

※ 제일저자(First Author) : 기안호
접수일:2011년 01월 29일, 수정일:2011년 05월 16일, 완료일:2011년 06월 24일
* 한양대학교 전자컴퓨터통신공학과
kyano@hanyang.ac.kr
** 한양대학교 컴퓨터공학부(교신저자)
이 논문은 2009년 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (2009-0073575)

는 전혀 다른 특성을 갖는 플래시 메모리 기반의 저장장치들이 계속해서 시장에 진입하고 있다[2]. 초창기의 저 용량 휴대용 저장장치에서 벗어나 현재는 하드디스크의 완전한 대체제인 SSD(Solid-State Drive)가 이미 우리 주변에서 흔하게 볼 수 있는 저장장치가 되었다. 그에 따라 여러 파일시스템들이 새로운 플래시 메모리 기반의 저장장치에 특화된 성능을 내기 위해 다듬어지고 개발되고 있다.

이처럼 새로운 매체에만 초점이 맞춰진 파일시스템들이 등장하는 동안 고전적인 파일시스템의 상위 인터페이스에도 변화가 요구되기 시작하였다. 사용자 데이터의 기하급수적인 증가에 따라 검색의 중요성이 강조되고 있으며, 웹을 시작으로 시맨틱 검색이 활성화되면서 파일들 역시 시맨틱 검색이 중요하게 떠오르기 시작하였다[3]. 특히 시맨틱 검색의 성능과 질을 향상시키기 위해 파일시스템 차원에서 자체적으로 보조를 맞추기 위한 노력들이 나타나고 있다.

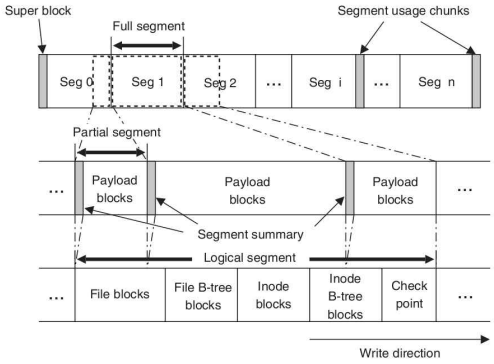
하지만 이 두 가지 새롭게 떠오르고 있는 사항에 대해 모두 고려해서 접근하고 있는 파일시스템은 아직까지 연구가 되고 있지 않다. 각각의 경우 적용되는 계층이 다르기 때문에 이를 통합해서 생각하고자 하는 시도가 없었기 때문이다. 하지만 아직 주목받지 못한 일부 SSD의 특성을 이용하면 시맨틱 검색을 위한 메타데이터를 유지하기에 적합한 로그기반 파일시스템을 설계할 수 있다. 본 논문에서는 이러한 특성을 이용하는 SSD에서의 시맨틱 파일 검색을 위한 확장된 속성 제공의 로그기반 파일시스템을 제안한다.

2. 관련 연구

2.1 로그기반 파일시스템

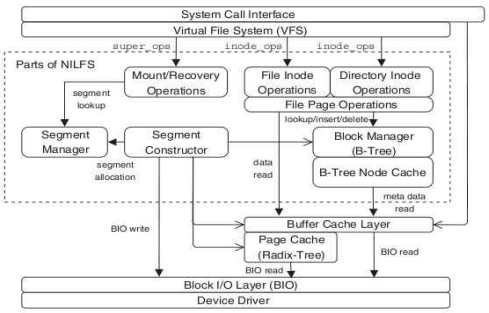
2.1.1 NILFS

NILFS는 리눅스에서 로그기반 파일시스템을 구현한 파일시스템이다[4]. NILFS는 고전적인 로그기반 파일시스템에서 지원하지 않는 몇 가지 기능을 추가적으로 지원하고 있다. 첫 번째로는 검사지점(checkpoint)을 사용하고 있는 점이다. 파일시스템이 손상된 후 빠른 복구를 하기 위해 DBMS에서 사용되는 검사지점 개념을 차용해서 사용하고 있다. 또, NILFS는 사용 중에 바로 정의 가능한 온라인 스냅샷을 지원하고 있다. 이는 시스템 중단 없는 백업이 가능하게 해준다.



(그림 1) NILFS의 디스크 구조

그림 1은 NILFS의 디스크 구조를 보여주고 있다. 디스크의 가장 앞에는 슈퍼블록이 위치하고 그 뒤에 데이터 세그먼트들이 위치한다. 슈퍼블록을 위한 별도의 특별한 세그먼트는 사용하지 않는다. 각각의 세그먼트들은 부분적인 세그먼트(partial segment)들로 구성되어 있고 연속된 부분적인 세그먼트들은 내부적으로 별도의 논리적 세그먼트로 구성된다. 이처럼 구성된 이유는 검사지점을 사용하다 보면 각각의 복구 단위는 디스크상의 세그먼트로 표현할 수 없는 크기의 데이터들은 하나의 단위로 다뤄야 하는 일이 발생하기 때문이다. 각각의 복구 단위는 논리적 세그먼트로 표현되고 이는 부분적인 세그먼트로 쪼개져서 실제 디스크에 저장된다. 세그먼트를 부분적인 세그먼트로 나눈 것은 무조건 세그먼트 단위로만 쓰다 보면 피할 수 없는 단편화 문제를 해결하기 위함이다.

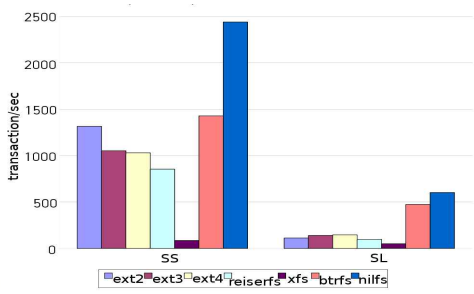


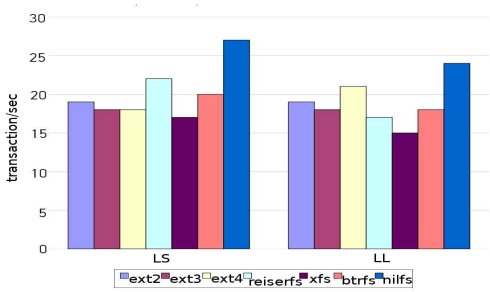
(그림 2) NILFS의 파일시스템 구조

그림 2에서 보면 NILFS의 경우 다른 리눅스 파일시스템들과 크게 3 부분에서 차이가 있다는 것을 알 수 있다. 먼저 세그먼트 관리자(Segment Manager)는 디스크상의 세그먼트 할당과 가비지 컬렉션에 해당하는 클리너(cleaner)의 동작을 책임진다. 세그먼트 생성자(Segment Constructor)는 디스크상의 블록들을 관리하며 부분적인 세그먼트들을 생성한다. 마지막으로 블록 관리자(Block manager)는 B-Tree를 통해 블록의 사용가능 여부 등을 관리한다.

2.1.2 SSD 상에서의 로그기반 파일시스템의 성능

SSD를 포함한 플래시 메모리 기반의 저장장치에서는 임의 위치 기록보다는 순차 기록이 훨씬 나은 성능을 보인다는 것은 익히 알려졌다[5][6]. 그러한 점을 고려해 볼 때 모든 쓰기 작업이 순차 기록으로 이루어지는[7] 로그기반 파일시스템은 SSD에서 훨씬 나은 성능을 보일 수 밖에 없다. [5]에서 발표된 결과에 따르면 NILFS의 경우 로그기반이 아닌 다른 파일시스템과 비교해서 월등한 성능을 보인다(그림 3).





(그림 3) 파일시스템별 SSD에서의 PostMark 결과
(上) 작은 파일 크기, (下) 큰 파일 크기

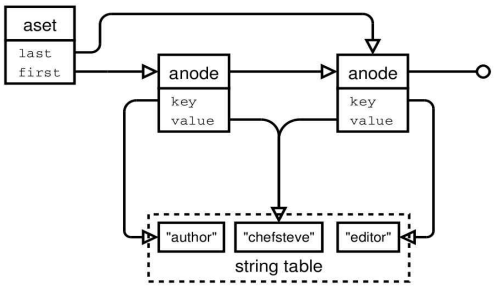
2.2 시맨틱 파일 검색

시맨틱 검색을 지원하는 파일시스템에 대한 연구는 1980년대 후반 데이터베이스 기반 파일시스템에 연구가 시작된 이후 조금씩 이루어져 왔다[6][8]. 그러다 웹의 등장 이후 문서의 수가 엄청나게 늘어나면서 효과적인 검색을 위해 시맨틱 웹/검색의 중요성이 강조되면서[9], 최근 들어 다시 강조가 되고 있는 분야이다.

본 논문에서는 이러한 연구 결과들 중에서 역시 최근 관심 받고 있는 스토리지 클래스 메모리(Storage Class Memories)에 시맨틱 검색을 접합하고자 한 LiFS[10]에 대해서 살펴본다.

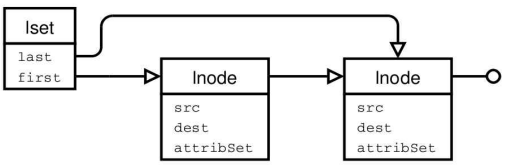
LiFS의 가장 큰 특징은 시맨틱 검색을 위해 증가된 메타데이터를 NVRAM을 사용하여 관리하겠다는 점이다. 그리고 또 하나, 모든 파일들 사이에는 파일시스템 밖으로 드러나지 않는 링크를 이용하여 연결시킨다는 점이다. 이런 링크를 사용하는 이유는 의미기반의 디렉터리 구조를 쉽게 만들고 관리하기 위함이다. 예를 들어 사용자 A가 소유하고 있는 음악 파일 b.mp3의 경우, 음악 파일을 모은 Music/ 디렉터리의 목록에도 존재를 해야 하고, 사용자 A의 파일을 모은 Users/A/ 디렉터리에도 존재할 수 있도록 해주는 방식이다. 기존 유닉스의 하드/심볼릭 링크를 매우 적극적으로 확장한 것이다.

그리고 크기가 커진 메타데이터의 크기를 최대한 억제하기 위해 문자열 표(string table)을 사용한다. 문자열 표와 속성 사이의 연결은 그림 4에 표현되어 있다.

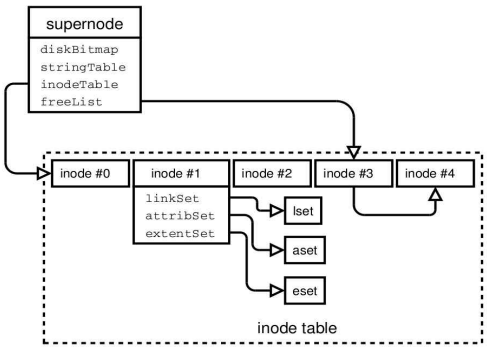


(그림 4) LiFS의 문자열 표

LiFS는 메타데이터의 저장을 바이트 단위로 접근이 가능한 NVRAM에 저장한다고 가정하였기 때문에 전체적으로 주소 포인터를 통한 파일과 파일 사이, 파일과 메타데이터 속성 사이, 메타데이터 속성과 문자열 표 사이에 링크가 가능하다.



(그림 5) LiFS의 파일사이의 연결 구조



(그림 6) LiFS의 Inode 연결 구조

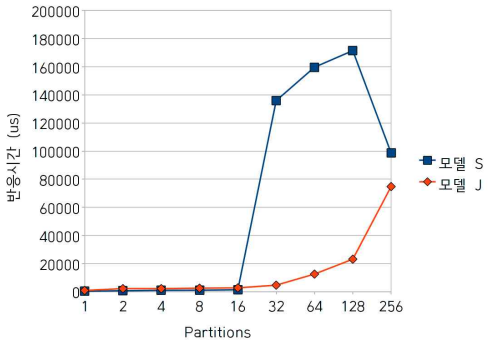
3. SSD를 위한 시맨틱 파일 검색

3.1 로그 구조를 가지는 저장장치 구조 설계

3.1.1 다중 로깅 지점

SSD에서는 하드디스크만 고려했던 기존의 로그기반 파일시스템과는 다른 다른 방식의 접근이 가능하다. 하나의 로깅 지점만을 유지하는 것이 아닌, 동시에 여러 로깅 지점을 유지하는 방식이 그것이다. 일반적으로 로그기반 파일시스템은 디스크를 크게 하나의 로깅 지점만을 갖는 저장장치로 생각하고 동작하였다. 하지만 SSD의 잘 알려지지 않은 특성을 이용하면 여러 로깅 지점의 동시 기록이 가능하다[11].

SSD는 내부적으로 FTL에서 데이터의 정리를 위해 로그 블록을 사용한다. 그래서 쓰기가 발생할 경우, 바로 해당하는 위치를 찾아서 데이터를 기록하는 것이 아니고, 일차적으로 로그 블록에 기록을 하고 이 데이터들을 주기적으로 정리하여 실제 위치에 기록을 하게 된다. 이 때 사용되어지는 로그 블록의 개수에 따라 SSD에 동시에 여러 순차 접근 데이터를 기록하여도 그 성능이 떨어지지 않게 된다. 이는 특히 현재 시장에서 가장 널리 쓰이고 있는 블록 단위 사상을 사용하는 SSD에서 그 이득이 크게 된다.



(그림 7) SSD에서의 다중 로깅 지점

그림 7에서 확인해 보면 시장에서 유통 중인 SSD의 경우, 16개의 동시 쓰기 작업에 대해서 그 반응 속도가 떨어지지 않음을 확인할 수 있다. 이 특성을 이용하면 순차 기록 작업인 로그 기록을 동시에 여러 지점에 할 수 있게 된다. 다중 로깅 지점은 로그기반 파일시스템에서 큰 효과를 낼 수 있다. 일반적으로 로그기반 파일시스템은 순차 기록만을 사용하기 때문에 SSD를 포함하여 디스크에서 큰 성능의 이득을 낼 수 있을 것이라 여겨지지만 실제로는 그렇지 못했다.

이 성능 하락의 병목 현상으로 지목되는 부분은 바로 가비지 컬렉션이다. 의미가 없어진 데이터를 계속 유지하고 있기 때문에 디스크 공간의 낭비가 생기게 되고, 이를 해결하기 위해 주기적으로 의미가 없어진 데이터를 정리하는 과정이 필요하게 된 것이다. 하지만 이 의미가 없어진 데이터를 정리하는 가비지 컬렉션 과정의 경우 어떤 데이터부터 정리해야 할 지 우선순위를 판단하는 비싼 연산을 행하게 된다. 또 데이터들이 참조되는 주기에 따라 Hot/Cold 속성이 나뉘게 되는데 이 속성에 따라 서로 다른 정책이 필요하지만 그렇게 할 수 없어 협상 점을 찾다 보니 최상의 가비지 컬렉션 정책을 사용하기 어렵다.

그러나 다중 로깅 지점을 사용하면, 각각의 지점에 서로 다른 속성을 갖는, 즉 Hot/Cold 속성에 따라 서로 분리하여 저장할 수 있기 때문에 각각의 속성에 따라 다른 가비지 컬렉션 정책을 사용할 수 있게 되고, 우선적으로 정리할 데이터를 찾기 위한 비싼 연산의 과정이 줄어든다.

이 특성을 이용하여 모두 3개의 로깅 지점을 사용하는 로그기반 파일시스템을 설계하였다. 각각 로깅 지점들은 파일시스템 메타데이터, Hot 속성의 파일데이터, Cold 속성의 파일데이터에 대해 로그를 기록하도록 하였다. 우선적으로 메타데이터는 다른 파일의 데이터보다 더 Hot 속성을 가지기 때문에 이를 하나의 로그로 기록하도록 하였다.

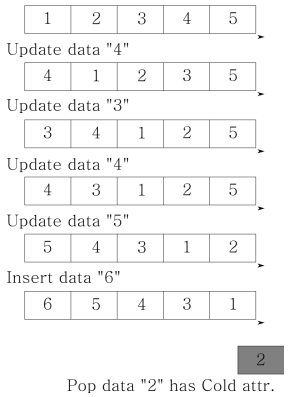
3.1.2 FIFO 큐를 이용한 Hot/Cold 분리 정책

파일시스템 메타데이터의 경우에는 전체적으로 Hot 속성을 가지기 때문에 따로 고민을 할 필요가 없었지만, 일반적인 파일데이터의 경우에는 그 파일의 특성에 따라 다른 속성을 띄기 마련이다. 그에 따라 파일데이터의 Hot/Cold 분리를 위하여 여기서는 FIFO 큐를 이용하기로 하였다.

FIFO 큐를 이용한 Hot/Cold 분리 정책은 다음과 같은 과정을 거친다[12].

1. 새로 기록되는 데이터는 큐의 가장 마지막 위치에 삽입된다.
2. 큐에 이미 들어온 데이터가 새로 갱신이 되면 큐에서 쫓겨나서 다시 제일 뒤에 삽입된다.
3. 이 과정을 반복적으로 거치게 되면 큐가 가득 차서 내보내지는 데이터는 Cold 속성을 띄게 되고 큐 내부에는 Hot 속성을 띄는 데이터들만 계속 남아있게 된다.

이를 그림으로 나타내면 아래 그림 8과 같다.



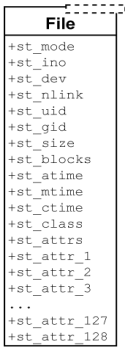
(그림 8) FIFO 큐를 사용하는 Hot/Cold 분리 과정

큐에서 내보내지는 파일데이터는 Cold 속성을 가지게 되므로 Cold 파일데이터를 기록하는 로그에 쓰게 되고, 데이터 영속성을 위해 주기적으로 큐에 있는 데이터들은 Hot 파일데이터를 기록하는 로그에 쓰게 된다.

3.2 시맨틱 파일 검색을 위한 메타데이터 설계

3.2.1 메타데이터 구조

시맨틱 검색을 위해서는 기존의 POSIX에 정의된 메타데이터 구조로는 부족하다. 파일의 의미데이터에 대해서 메타데이터에 기록을 해둬야지만 검색이 가능하기 때문이다. 이를 위해서 기존 POSIX 파일 속성에 추가적인 속성을 더하였다. 이 때 추가 속성은 파일 형식에 따라 각기 다른 미리 정의된 속성을 가지도록 하였다.



(그림 9) 파일시스템 메타데이터의 기본 구조

그림 9에서 보이는 각각의 속성은 다음을 의미한다.

<표 1> 파일시스템 메타데이터의 기본 구조 설명

속성	설명
st_mode	POSIX 파일 모드
st_ino	파일의 Inode 번호
st_dev	파일이 존재하는 블록 디바이스
st_nlink	파일에 대한 하드링크의 개수
st_uid	파일의 소유자 정보
st_gid	파일의 소유자 그룹 정보
st_size	파일의 크기
st_blocks	파일이 가지는 블록 개수
st_atime	파일의 최종 접근 시간
st_mtime	파일데이터의 최종 수정 시간
st_ctime	파일 메타데이터의 최종 수정 시간
st_class	**파일의 형식**
st_attrs	**추가된 속성의 개수**
st_attr_?	**추가된 속성들**

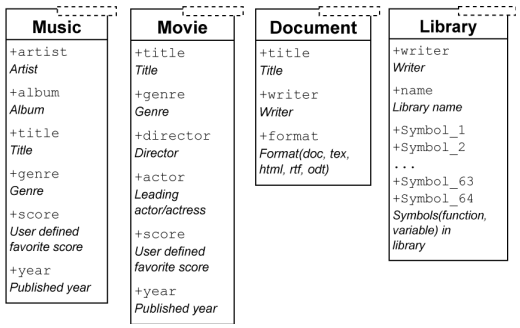
표 1에서 굵은 글씨로 표현된 속성들은 기존의 POSIX 규격에 정의되지 않은, 이 파일시스템에서 새롭게 추가한 속성들이다. 새롭게 추가된 속성에 대해 더 자세히 살펴보면 다음과 같다.

1. st_class : 파일의 형식을 나타낸다. 여기에는 이 파일시스템이 기본적으로 검색 가능한 음악(mp3, flac, ogg, aac), 영상(avi, mkv, mp4, flv), 문서(doc, tex, html, rtf, odt), 라이브러리(la, so)를 나타낸다. 그 외의 파일에 대해서는 확장자가 들어간다. 사용자 정의 태그가 있을 경우 그러한 태그가 있음을 알리는 값(-1)을 넣는다.

2. st_attrs : 추가된 속성의 개수가 들어간다. 검색을 위한 속성은 최대 128개까지 추가될 수 있고, 파일의 형식에 의해 기본적으로 정의되는 속성들 외에도 사용자가 정의한 속성들이 더 들어갈 수 있기 때문에 전체 개수를 별도로 기록한다.

3. st_attr_? : 추가된 속성들의 값이 들어간다.

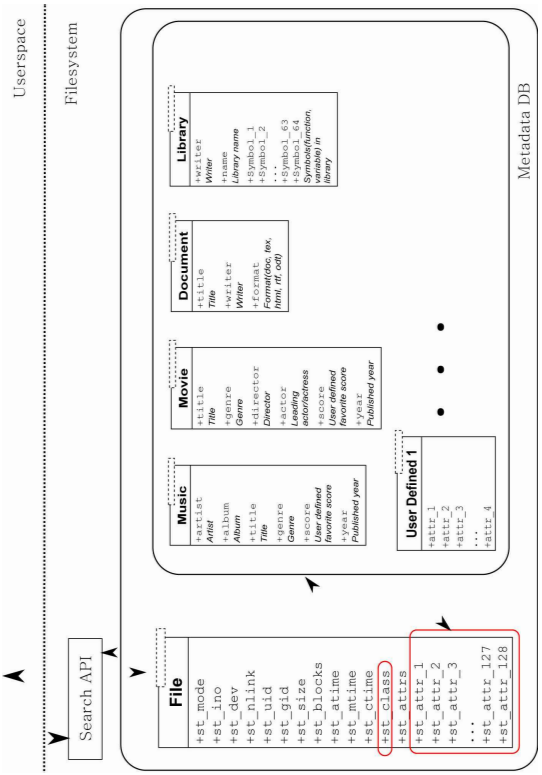
미리 정의된 음악, 영상, 문서, 라이브러리의 경우 다음과 같은 속성을 가진다. 현재 구현에는 4가지 형태에 대해서만 지원하고 있지만, 사용자 플러그인 형태로 필요 시 새로운 형태의 추가가 가능하다.



(그림 10) 미리 정의된 파일에 대한 추가 속성

각각의 속성들에 대해 이름을 미리 기록할 경우 검색의 복잡성이 증가하기 때문에 각각의 속성들이 미리 정의된 원형을 별도로 제공하여 쓰도록 하고 있다. 실제 검색 시에는 다음과 같은 흐름을 가진다.

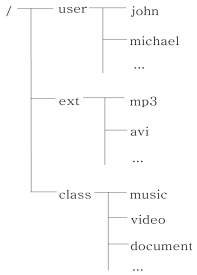
그림 11에서 보이듯이 사용자로부터 검색 요청이 들어오면 파일시스템은 메타데이터가 저장된 DB를 이용하여 검색을 한다. 메타데이터 DB는 기본 속성을 우선적으로 검색하고 그 곳에 정의된 파일의 형식에 맞는 추가 속성을 검색하여 st_attr_[1...128] 중에서 사용자가 요청한 속성이 몇 번째에 저장되어 있는 지를 확인한다. 그 후 해당하는 값이 입력된 파일을 반환해 줌으로써 검색이 종료된다. “User Defined ” 테이블은 사용자가 자신만의 태그를 정의하였을 경우 해당하는 추가 속성이 저장된다. 이 테이블을 통해 미리 정의되지 않은 속성의 추가를 더욱 유연하게 하였다.



(그림 11) 검색 시 동작의 흐름

3.2.2 메타데이터 검색을 위한 VFS 구조

매번 필요한 파일을 찾기 위해 사용자 프로그램을 이용하여 직접 검색을 하는 것은 사용자 편의성도 떨어지고 매번 검색을 수행하는 추가 비용이 발생하기 때문에 VFS 수준에서 잦은 검색에 대해서 미리 정의하여 탐색할 수 있도록 하였다. 파일의 형식, 소유자 등의 정보를 이용하여 디렉터리를 생성하고 해당 디렉터리에서 파일 목록을 확인하면 각각의 디렉터리가 의미하는 바에 맞는 파일들을 바로 찾을 수 있다. 파일시스템에 의해 생성되는 기본 디렉터리 구조는 다음과 같다.



(그림 12) 메타데이터 검색을 위한 기본 디렉터리 구조

최상위 디렉터리로는 파일의 소유자, 확장자, 형식에 대해서 생성된다. 소유자는 파일의 uid값에 기초해서 하위 디렉터리들이 생성되고 각각의 디렉터리에는 해당 사용자의 파일이 모두 검색된다. 확장자는 파일의 확장자별로 검색될 수 있도록 존재하는 모든 파일의 확장에 대해서 하위 디렉터리가 생성된다. 형식은 앞서 말한 st_class 값에 기초해서 생성되며 각각의 형식에 맞는 파일들이 하위 디렉터리에서 검색된다. 각각의 하위 디렉터리에서 추가 검색을 수행하면 전체 파일이 아닌 해당 디렉터리에 해당하는 파일들 중에서 사용자 요청에 맞는 파일들이 검색된다.

3.2.3 메타데이터 DB

이 장에서 설명한 파일시스템의 구현을 위해 FUSE[13]와 SQLite DB를 이용하였다. SQLite DB를 통해 메타데이터를 저장하고 SQLite DB로 인해 생성되는 파일을 미리 정의된 메타데이터를 위한 로깅 지점에 기록하였다. 메타데이터의 저장을 위해 DB를 사용한 이유는 DBMS에서 기본으로 제공해주는 트랜잭션 속성을 이용하여 파일시스템의 영속성 향상과 빠른 복구를 하기 위함이다. 특히 메타데이터의 검색은 그 구조가 유연할수록 성능이 떨어지기 마련인데, 성능 하락의 폭이 예상치를 넘어서지 않게 파일 인덱싱/검색 프로그램들에서 널리 사용된 DBMS인 SQLite DB를 사용하였다. 파일이 가지는 각각의 속성에 대해서는 SQLite DB가 제공하는 인덱스 기능을 이용하여, 소유자정보, 추가 속성에 대해서 인덱싱을 하였다. 이를 통해 빠른 검색을 가능하게 하였다.

미리 정의된 검색 디렉터리에 대해서는 마운

트 시점에 DBMS에 view를 생성한다.

4. 성능평가

4.1 실험 환경

실험은 AMD Phenom X3 8450 Triple-Core CPU에 8GB RAM이 장착된 PC에 설치된 리눅스 환경에서 이루어졌다. 그리고 리눅스 커널 2.6.34, FUSE 2.8.1, SQLite DB 3.6.23.1, Python 2.6.5, Python-pysqlite 2.6.0, Python-fuse 0.2.1 환경에서 파일시스템을 구현하였다. SSD는 SATA2 인터페이스에 연결된 32GB 용량의 A-DATA社의 S391 모델이 사용되었다.

파일샘플은 mp3, flac, ogg 오디오파일 799개, avi, mkv, mp4, flv 영상파일 423개, doc, tex, rtf, html, odt 문서 파일 389개를 사용하였다. 모든 실험은 5회의 실시 후 그 평균값을 취하였다.

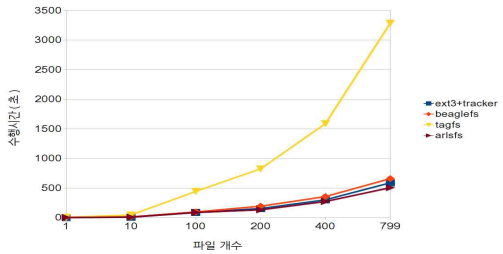
4.2 파일 생성과 인덱스 생성

현재 데스크탑 환경에서 시맨틱 파일 검색을 지원해 주는 방법으로는 별도의 전용 검색 프로그램을 이용하는 방법이 널리 쓰이고 있다. 그에 따라 기존의 알려진 몇몇 시맨틱 검색을 지원하는 파일시스템 외에 데스크탑용 파일 검색 프로그램을 포함하여 실험을 수행하였다. 비교 대상으로 삼은 파일시스템/프로그램은 다음과 같다.

<표 2> 실험 대상 파일시스템/프로그램

파일시스템/프로그램	설명
ext3 + tracker	기존 파일시스템인 ext3에 유닉스용 데스크탑 파일 인덱싱/검색 프로그램인 tracker를 사용.
beaglefs	기존 파일시스템 위에 유닉스용 데스크탑 파일 인덱싱/검색 프로그램인 beagle을 통해 인덱싱한 후 FUSE를 이용해 만들어진 의사(pseudo) 시맨틱 파일시스템.
tagfs	SQLite DB를 사용하는 FUSE로 구현된 시맨틱 파일시스템.
arlsfs	논문에서 제안한 파일시스템

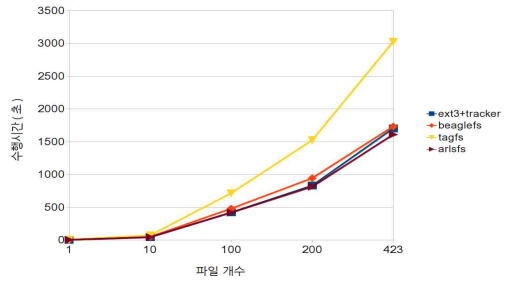
우선 미리 준비된 음악 파일을 복사하면서 복사에 걸리는 시간과 인덱싱에 걸리는 시간을 측정하였다. 음악 파일의 인덱싱에 사용되는 속성은 “가수, 노래 제목, 장르, 출시연도” 4가지 항목에 대해서 수행되도록 하였다. 총 799개의 음악 파일을 개수에 변화를 주면서 실험하였다.



(그림 13) 음악 파일에 대한 생성/인덱싱 속도 비교

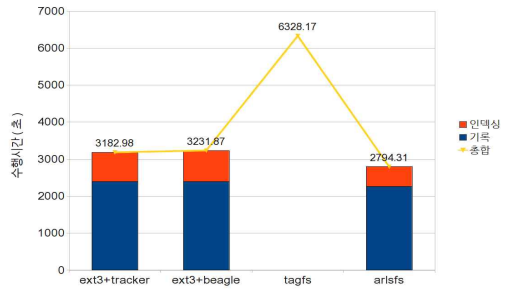
그림 13을 보면 음악 파일 생성/인덱싱 속도는 논문의 파일시스템이 다른 파일시스템/프로그램보다 빠르게 나왔다. beaglefs는 ext3 파일시스템 위에 구현된 것이나 beagle자체의 속도가 tracker보다 느려 ext3 + tracker보다 속도가 느리게 나왔다. tagfs는 다른 파일시스템/프로그램과 큰 차이로 느리게 나왔다. 이는 인덱싱보다 파일데이터 자체를 DB에 저장하는 과정에서 발생하는 추가비용과 함께 최적화되지 못한 파일시스템의 내부적인 문제로 보인다.* 전체 799개의 파일을 생성하는 경우 논문의 파일시스템은 ext3 + tracker보다 14% 더 빠르게 나왔다.

* tagfs는 현재 성능보단 시맨틱 검색의 구현에 중점을 두고 개발되고 있다.



(그림 14) 동영상 파일에 대한 생성/인덱싱 속도 비교

그림 14에서 볼 수 있듯 동영상 파일에서는 파일시스템/프로그램 사이에 차이가 더 좁아졌다. 이는 상대적으로 파일 하나의 크기가 커지게 됨에 따라 대부분의 쓰기 작업이 순차 쓰기로 이뤄지면서 로그기반 파일시스템의 장점이 희석된 때문이다. 논문의 파일시스템의 경우 ext3 + tracker와 비교해 보았을 때, 파일의 개수가 100개 미만일 경우에는 적은 차이로 더 느렸으나, 파일의 개수가 증가함에 따라 더 빠른 성능을 보여주었다. 423개의 파일을 모두 생성하는 경우에는 논문의 파일시스템은 ext3 + tracker보다 5% 더 빠르게 나왔다.



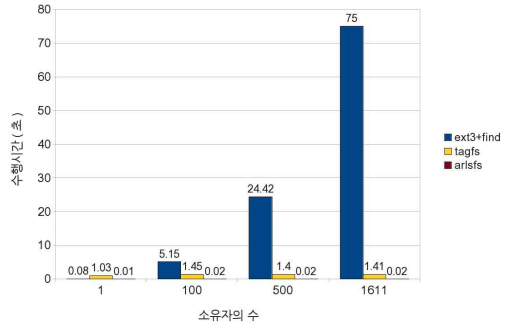
(그림 15) 인덱싱 속도 비교

그림 15는 준비된 음악 파일, 동영상 파일, 문서 파일, 총 1611개의 파일을 한 번에 기록하고 전체 인덱싱을 하는 과정에서의 성능을 측정하였다. beaglefs의 경우 ext3 파일시스템 위에 beagle을 통해 인덱싱을 하고 이를 다시 파일시스템의 형태로 보여주기 때문에 파일데이터 기록 시간은 ext3로 측정하고 beagle을 통한 인덱싱 시간만을 별도로 측정하였다. 그리고 tagfs의 경우에는 시간을 별도로 측정할 수 없기 때문에 전체 시간만을 측정하였다. 논문의 파일시스템 역시 인덱싱과 데이터 기록이 동시에 일어나기 때문에 코드를 수정하여 시간을 별도로 기록하도록 하였다. 파일의 개수가 많아지고 종류가 다양해지니 tracker나 beagle의 인덱싱 속도가 크게 나빠진 것을 확인할 수 있다. 인덱싱의 경우 beagle보다 36%, tracker보다 32% 더 빠른 성능을 보여준다. 파일데이터 기록 시간의 경우 ext3보다 5% 더 빠른 성능을 보여준다. 이 실험결과를 통해 논문의 파일시스템이 로그기반 파일시스템이 주는 이점을 그대로 가져오면서 확장된 속성의 메타데이터 인덱싱에 있어서도 성능이 우수함을 확인할 수 있다.

4.3 파일과 디렉터리 탐색

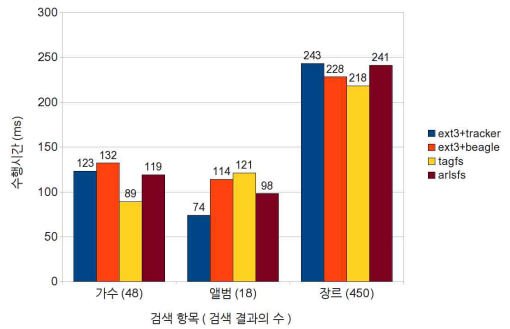
이번에는 미리 준비된 1611개의 파일에 대해서 미리 인덱싱까지 끝마친 상태에서 탐색할 때 걸리는 시간을 측정하였다. 먼저 기존의 POSIX 정의 속성인 파일의 소유자별로 파일의 목록을 검색하는 실험을 수행하였다. tracker와 beagle의 경우 파일소유자에 대한 검색을 지원하지 않아 유닉스 프로그램인 find를 이용하였다.

그림 16의 X축은 전체 소유자의 수이다. 모든 파일이 한 사용자의 소유로 되어있을 경우에는 tagfs를 제외하고는 서로 성능이 비슷하였다. 그러나 파일을 균등하게 100명의 사용자, 500명의 사용자, 마지막으로 모든 파일이 서로 다른 사용자에게 소유되었다고 실험환경을 설정한 경우에는 find 명령을 사용하는 경우 성능이 급격히 나빠졌다.



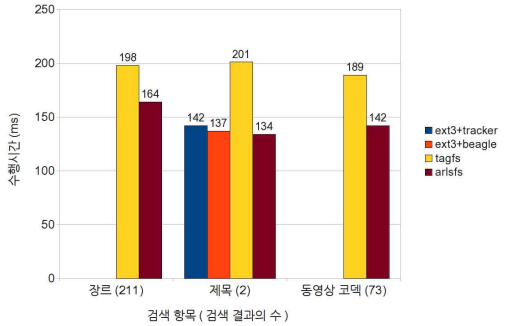
(그림 16) 파일소유자별 목록 검색 속도 비교

이는 매번 메타데이터를 검색해야 하는 파일시스템의 구조적인 단점 때문이다. 그러나 시맨틱 검색을 위해 별도의 메타데이터 데이터베이스를 유지하는 다른 파일시스템들은 한 번의 검색으로 분류가 가능하기 때문에 성능 하락이 발생하지 않았다. 특히 논문의 시스템의 경우 파일 소유자에 대해서는 마운트 시점에 미리 생성된 view를 이용하기 때문에 검색에 추가비용이 거의 존재하지 않는다.



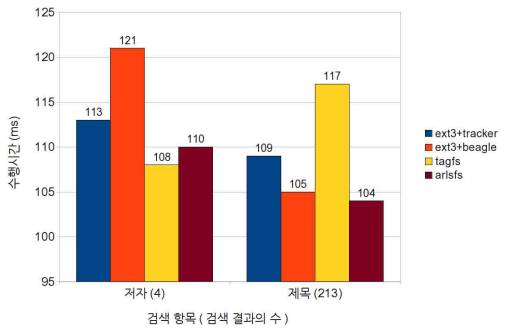
(그림 17) 음악 파일 검색

그림 17에서는 음악 파일에 대한 검색을 수행하였다. 각각 가수, 앨범, 장르에 대한 검색을 수행하였으며 그 결과로 48개, 18개, 450개의 파일을 찾아주는 검색이 되겠다. 파일시스템/프로그램에 따른 차이 없이 최대 40ms이내의 차이만을 보여줬다. 이는 이미 인덱싱이 된 파일들에 대해서는 검색에 따른 추가비용이 크지 않음을 의미한다.



(그림 18) 영상파일 검색

영상파일의 경우에는 그림 18에서 볼 수 있듯, tracker와 beagle의 경우 내부 태그에 대한 검색을 제공하지 않기 때문에 제목에 대한 검색만 수행하였다. 사용자 태그에 대해서도 실험을 수행해 보기 위해 논문의 파일시스템에서 기본적으로 제공하지 않는 동영상 코덱에 대해서 사용자 태그로 입력한 후 실험하였다. 음악 파일과 비슷하게 서로 간 차이가 크지 않은 결과를 보여줬다.



(그림 19) 문서파일 검색

그림 19는 문서파일에 대한 검색 결과를 보여주고 있다. 문서파일 역시 앞선 실험과 큰 차이가 없는 결과를 보여주고 있다.

실험결과를 통해서 논문의 파일시스템은 현재 쓰이는 파일시스템/프로그램보다 빠른 인덱싱 속도를 가지면서 더 많은 검색 항목을 제공하는 파일시스템임을 볼 수 있다.

5. 결론

본 논문에서는 SSD에서 최적의 성능을 낸다고 알려진 로그기반 파일시스템과 최근 관심을 받고 있는 시맨틱 검색 기능을 제공하는 파일시스템을 하나로 묶어서 SSD에 최적화된 시맨틱 검색 파일시스템을 제안했다. SSD의 멀티 로깅 지점 지원 특성을 활용하여 시맨틱 검색 파일시스템을 설계하면 성능에서 큰 이득을 얻을 수 있음을 실험을 통해 보였다. 또한 기존에 저장 매체와는 별개로 생각되어지던 시맨틱 검색을 위한 파일시스템 설계를 저장 매체의 특성에 맞게 개선해서 더 나은 효과를 보일 수 있다는 것을 보여줬다.

본 논문에서는 제안한 시스템을 실제 시스템에 구현하여 기존 시스템과의 성능을 비교했다. 제안한 시스템은 기존 시스템에서 사용자 영역 프로그램으로 하던 작업을 파일시스템 수준으로 끌어내려 추가적인 인덱싱 작업이 필요 없었다. 또한 추가적인 검색 프로그램을 사용하는 것보다 인덱싱에 걸리는 시간이 짧다는 것도 보여줬다. 그리고 기존의 프로그램들보다 더 많은 항목에 대한 검색을 지원하면서 검색에 걸리는 시간 역시 비슷하거나 더 짧아 전체적으로 크게 향상된 성능을 보였다. 특히 기존의 사용자 영역 프로그램 기반의 검색 작업은 각각의 프로그램마다 고유의 데이터베이스를 별도로 운영하여 필요에 따라 중복된 인덱스 생성이 필요하였지만 파일시스템에서 모든 작업이 이미 이루어졌기 때문에 단순한 디렉터리 리스팅만으로 사용자는 원하는 파일에 쉽게 접근이 가능했다. 마지막으로 다른 시맨틱 검색을 위한 파일시스템들이 사용자에게 직접 태그 형식의 속성 추가를 필요로 하는 반면, 논문의 파일시스템은 기존의 tracker, beagle보다 더 많은 수준에서 미리 정의된 파일에 대해서 자동으로 속성을 찾아 인덱싱을 지원해주고 있다.

본 논문에서는 그동안 서로 다른 계층으로만 여겨지던 두 가지 파일시스템에 대한 접근법에 대해서 매체의 특성이 시맨틱 검색과 같은 상위 계층에서의 작업에 대해서도 도움을 줄 수 있음을 보여줌에 의의가 있다. 앞으로는 페이지 단위 사상과 같은 다른 매체 특성의 SSD에 대해서도 의미가 있는지와 점차 발전해 가는 시맨틱 검색의 패러다임에서도 계속 매체의 특성을 고려하는 것이 가능한가에 대한 연구가 필요할 것이다.

참 고 문 헌

[1] ThomasM.Coughlin. Digital Storage for Professional Media and Entertainment. 2009. Coughlin Associates.

[2] DouglasDixon. The Flash Storage Revolution. Manifest Technology Website. Apr. 2009. Princeton University.

[3] T.Berners-Lee and E.Miller. The semantic web lifts off. ERCIM News. Oct. 2002.

[4] RyusukeKonishi, YoshijiAmagai, KojiSato, HisashiHifumi, SeijiKihara, and SatoshiMoriai. The Linux implementation of a log-structured file system. SIGOPS Oper. Syst. Rev.. 0163-5980. 3. 102-107. 2006. ACM. New York, NY, USA.

[5] DongjunShin. SSD. LSF '08: 2008 Linux Storage &Filesystem Workshop. February 25-26, 2008. ACM. San Jose, CA.

[6] DavidK.Gifford, PierreJouvelot, MarkA.Sheldon, and JamesW.O'toole. Semantic File Systems. IN 13TH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES. 16-25. 1991. ACM.

[7] MendelRosenblum and JohnK.Ousterhout. The design and implementation of a log-structured file system. ACM Trans. Comput. Syst.. 0734-2071. 1. 26-52. 1992. ACM. New York, NY, USA.

[8] MichaelA.Olson and MichaelA. The Design and Implementation of the Inversion File System. 1993.

[9] TimBernersLee. Semantic web roadmap. http://w3.org/DesignIssues/Semantic.html. Sept. 1998.

[10] SashaAmes, NikhilBobb, KevinM.Greenan, OwenS.Hofmann, MarkW.Storer, CarlosMaltzahn, EthanL.Miller, and ScottA.Brandt. LiFS: An attribute-rich file system for storage class memories. In Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies (College Park, MD). 2006. IEEE.

[11] LucBouganim, BjörnÞórJónsson, and PhilippeBonnet. uFLIP: Understanding Flash IO Patterns. CIDR 2009, FOURTH BIENNIAL CONFERENCE ON INNOVATIVE DATA SYSTEMS RESEARCH. 2009. CIDR.

[12] NamyoonWoo and JeongeunKim. An Efficient Cleaning Policy using Hot and Cold Data Separation for Flash Memory. The 4th Samsung Tech Conference. Oct., 2007. Samsung Electronics. Co.,Ltd..
[13] Filesystem in UserSpace. http://fuse.sourceforge.net. 2008.

기 안 호

2008년 : 한양대학교 컴퓨터교육과 (학사)
2010년 : 한양대학교 전자컴퓨터통신공학과 (석사)

현 재: 한양대학교 전자컴퓨터통신공학과 (박사)
관심분야 : 플래시메모리 기반 저장 시스템, 파일시스템, 가상화, GPGPU

강 수 용

1996년 : 서울대학교 수학과(학사)
1998년 : 서울대학교 전산과학과 (석사)
2002년 : 서울대학교 전기컴퓨터공학부 (박사)

2003년~현 재: 한양대학교 컴퓨터공학부 교수
관심분야 : 멀티미디어 시스템, 분산시스템, 플래시메모리 기반 저장 시스템 등