선별적 데이터 판별에 의한 압축 알고리즘 효율 개선에 관한 연구

장승주*

A Study for Efficiency Improvement of Compression Algorithm with
Selective Data Distinction

Seung Ju Jang*

이 논문은 2012학년도 동의대학교 교내연구비에 의해 연구되었음(과제번호:2012AA177)

요 약

본 논문은 데이터 압축 효율 향상을 위하여 데이터에 대해서 무조건적인 압축을 시행하는 것이 아니라 선별적으로 데이터를 판별하여 압축을 하도록 한다. 선별적 데이터 판별을 통해서 압축 여부를 판단하게 된다. 이렇게 함으로써 압축 효율이 좋지 않은 경우에 대한 회피를 통해서 불필요한 압축을 하지 않을 수 있도록 한다. 불필요한 연산을 줄임으로써 압축 알고리즘의 성능 향상을 꾀할 수 있다. 특히, 이미 압축 알고리즘이 적용이 된 데이터의 경우에는 불필요한 압축을 하지 않도록 한다. 본 논문에서 제안하는 기능에 대해 실제 구현하고, 구현된 내용에 대해서 실험을 수행하였다. 본 논문에서 제시한 내용에 대해서 실험한 결과 정상적인 동작이 됨을 확인할 수 있었다.

ABSTRACT

This paper suggests to compress data selectively for improvement of data compression efficiency, not to perform unconditional compression on data. Whether to compress or not is determined by selective data distinction. By doing so, we can avoid unnecessary compression in the case of low compression efficiency. Cutting down the unnecessary compression, we can improve the performance of the pre-compression algorithm. Especially, the data algorithm which was already compressed could not be compressed efficiently in many cases, even if apply compression algorithm again. Even in these cases, we don't have to compress data unnecessarily. We implemented the proposed function actually and performed experiments with implementation. The experimental results showed normal operation.

키워드

선택적 압축 알고리즘, 압축 효율, 압축 알고리즘, 리눅스 시스템

key word

Selective Compression Algorithm, Efficient Compression, Compression Algorithm, Linux System

* 정회원 : 동의대학교 컴퓨터공학과 교수(sjjang@deu.ac.kr)

접수일자 : 2012. 12. 13

심사완료일자 : 2012. 12. 26

Open Access <http://dx.doi.org/10.6109/jkiice.2013.17.4.902>

©This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License(<http://creativecommons.org/licenses/by-nc/3.0/>) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.
Copyright © The Korea Institute of Information and Communication Engineering.

I. 서 론

현대 사회에서 컴퓨터는 거의 모든 분야에서 사용하고 있으며, 사용 범위는 점차 넓어져가고 있다. 사회 전체에서 컴퓨터의 사용 범위가 계속해서 늘어나고 컴퓨터 시스템이 매우 중요한 역할을 수행하고 있는 추세다. 특히, 컴퓨터 시스템에 장애가 발생하거나 중요한 정보를 보호하기 위하여 안정성과 보안이 가장 중요시 되고 있다. 시스템 안정성에 문제가 생길 경우 파일 시스템 전체의 동작이 중단 되고 작업 중이었던 데이터가 손실 되거나 기존 파일 시스템의 데이터까지 복구가 불가능하게 되는 상황이 발생 할 수 있다.

컴퓨터 사용량이 증가함으로써 파일 시스템의 안정성도 중요시 되고 있다. 파일 시스템이란, 파일의 내용과 그 파일과 관련된 데이터 즉, 메타데이터들을 유지하고 관리하는 체계이다. 리눅스는 표준 파일 시스템으로 EXT 파일 시스템, EXT2 파일 시스템으로 변천해왔다. EXT3 파일 시스템은 매우 안정적이고 치명적인 문제점도 없으며, EXT2 파일 시스템에서 EXT3 파일 시스템으로의 변경도 쉽고 간단하여 일반적으로 많이 사용 되고 있는 저널링 파일 시스템이다. EXT3 파일 시스템은 기본적으로 ordered mode를 사용한다. Ordered mode는 메타데이터를 수정하기 전, 저널링 공간에 수정하고자 하는 메타데이터를 기록하고 저장장치에 데이터를 쓴 다음 저널에 있는 메타데이터에 대한 정보를 저장장치에 옮긴다. 저장장치에 메타데이터보다 데이터를 먼저 써서 데이터를 저널링 하지 않고도 데이터의 손상을 막을 수 있다. 하지만 한번의 쓰기 동작을 위해 실제로 두 번의 쓰기 동작이 필요하고, 추가적인 디스크 공간이 필요하다.

하지만 저널링 파일 시스템은 로그를 저널이라는 공간에 기록하기 위해 추가적으로 쓰기 동작을 필요로 하고, 로그를 기록하기 위한 저널이라는 추가적인 공간까지 사용하고 있다[3,7]. 본 논문에서는 로그를 기록하는 저널공간으로 인해 사용되는 저장장치의 효율적인 활용을 위한 기능을 설계 및 구현한다.

본 논문에서는 압축 저장 기법을 적용하여 효율적인 저장장치 공간 관리와 쓰기 속도를 향상시키는 EXT3 파일 시스템의 ordered mode의 개선을 제안한다. 본 논문의 구성은 다음과 같다. 2장에서는 본 논문과 관련된 연구를 설명하고, 3장에서는 EXT3 파일 시스템의 구성에 대

해 기술하였다. 4장에서는 개선된 저널링 파일 시스템의 설계에 대해 설명하고 5장에서는 본 연구의 결론으로 구성한다.

II. 관련연구

리눅스 커널에서 디바이스 드라이버는 시스템 콜 인터페이스와 하드웨어 제어 사이에 위치하며 응용 프로그램이 디바이스 드라이버를 통해 각종 주변 하드웨어 장치에 접근할 수 있는 기능을 제공한다. 시스템에 새로운 디바이스를 장착한 후 디바이스 드라이버를 직접 커널에 포함한다. 디바이스 드라이버에는 캐릭터 디바이스 드라이버와 블록 디바이스 드라이버가 있으며 블록 디바이스 드라이버는 하드디스크처럼 내부에 파일 시스템을 가질 수 있는 디바이스 드라이버를 말하며, 임의 접근이 가능하고 블록 단위의 데이터 전송이 가능하다 [1,2,3].

그림 1. 리눅스 커널의 디바이스 드라이버
Fig. 1 Device Driver of the Linux Kernel

커널 모듈 프로그램이란 필요에 따라 커널에 load 하거나 unload 할 수 있는 특정한 기능을 수행하는 프로그램이다. 커널의 리빌드 과정과 시스템의 리셋 과정을 거치면 엄청난 시스템 오버헤드를 발생하는데 커널 모듈 프로그램을 이용하면 모듈 개발 시간을 단축시킬 수 있으며 커널의 기능을 확장하기 위해 커널 재 컴파일 과정을 생략할 수 있다.

저널링 파일 시스템에서 저널은 파일 시스템에 반영될 수정 사항을 버퍼에 로그로 남겨두었다가 주기적으로 파일 시스템에 커밋된다. 비정상적 종료 발생하면, 저널은 저장되지 않은 데이터를 복구하기 위한 검사 지점으로 사용되고 파일시스템의 메타 데이터의 손상을 막아준다 [4,5,6,7].

EXT3 파일 시스템에서 저널링은 세 가지로 나뉘어진다. **writeback mode**는 메타 데이터만 저널에 기록되며, 데이터는 저장장치 해당 위치에 직접 기록된다. 이렇게 하면 파일 시스템 구조를 보존해 손상을 방지하지만, 메타데이터를 저장하고 나서 데이터를 기록하기 전에 시스템이 비정상적인 종료를 일으킬 때 데이터가 손상될 수 있다. 이런 문제를 해결하기 위해 **ordered mode**를 사용할 수 있다. **ordered mode**는 데이터를 기록한 다음에야 메타데이터를 저널에 기록한다. 이런 방식으로 복구 이후에 데이터와 파일 시스템 일관성을 보장한다. 마지막으로 **journal mode**는 메타데이터와 데이터가 모두 저널에 기록된다. **journal mode**는 파일 시스템 손상과 데이터 손실에 대해 리스크를 최소화 할 수 있지만 성능이 저하 된다. **XFS, ReiserFS** 등 다른 저널링 파일 시스템은 메타데이터를 수정하는 중 시스템에 문제가 발생하여 비정상적인 종료를 하게 되면 일관성을 유지하지 못하는 것과 다르게 **EXT3** 파일 시스템의 **ordered mode**는 파일시스템의 일관성을 유지시키는 안정성을 보여준다 [4,8,9,10,11,12].

III. 선택적 압축 알고리즘 설계

EXT3 파일 시스템에 저장장치의 효율적인 공간 활용과 쓰기 속도를 향상시키기 위해 파일에 압축 저장 기법을 적용한다. 파일의 압축 시 사용자 수준의 압축 파일과 비압축 파일을 식별한 후 비압축 파일만 압축 저장 기법을 적용한다. 여기서 사용자 수준의 압축 파일이란, **mp3, mpeg, jpeg, pdf** 파일등과 같은 멀티미디어 파일과 같이 사용자 수준에서 압축 기법이 적용된 파일을 말한다. 그림 2 는 논문에서 선택적 압축 알고리즘을 설계한 시스템 구조이다.

그림 2. 시스템 구조
Fig. 2 System Structure

아래 그림 3은 압축 프로그램의 동작 흐름이다. 모듈 프로그램은 리눅스 배포판인 **RedHat 9.0**을 사용하여 개발하였다. **RedHat 9.0**은 리눅스 커널 버전 **2.4.20-8**을 사용하며 **EXT3** 파일 시스템을 기본적으로 지원한다. 압축 프로그램은 **zlib**라이브러리의 **gzwrite(), gzopen()** 함수 등을 이용하여 파일처리와 에러 처리를 구현하였다.

그림 3. 압축 알고리즘 플로우 차트
Fig. 3 Compression Algorithm Flow Chart

커널 모듈 프로그램에서 압축 프로그램이 호출되면 파일을 access하게 되고 정상적으로 파일이 access되었을 경우 파일의 앞부분을 정해진 크기만큼 압축을 수행한다. 압축된 파일의 크기와 미리 지정된 임계값과의 크기를 비교 하여 압축률을 구한 뒤 파일의 압축 후 저장 또는 원본파일 저장을 판단하게 된다.

압축률의 임계값을 설정하여 파일이 임계값보다 낮은 압축률을 보일 경우 사용자 수준의 압축 파일로 판단하고 파일의 나머지 부분은 압축하지 않는다. 임계값보다 압축률이 높을 경우 파일의 나머지 부분도 압축을 수행하고 압축된 파일을 디스크에 쓰기 한다. 사용자 수준의 압축 파일과 같이 압축률이 낮은 파일의 경우 압축 수행 후 원본 크기와 비슷한 크기의 압축 파일이 생성되어 압축 효율이 떨어질 뿐만 아니라 파일 중복 압축에 따른 불필요한 프로세서 낭비와 쓰기 수행시간까지 늘어나게 된다.

높은 압축률을 보이는 파일의 경우, 데이터 압축 시 소모되는 시간과 압축된 데이터를 디스크에 기록하는 시간이 원본 파일을 디스크에 기록하는 시간보다 수행 시간이 단축 될 것이다. 반면, 사용자 수준의 파일과 같이 낮은 압축률을 보이는 파일은 원본 크기와 비슷한 크기의 압축 파일이 생성 되므로 불필요한 파일 압축에 따른 자원이 낭비가 되고 디스크에 기록하는 시간 또한 늘어날 것이다. 따라서 압축률이 낮은 파일과 압축률이 높은 파일을 식별하여 선택적으로 압축하는 기법이 요구 된다.

그림 4. 높은 압축률을 보이는 파일과 낮은 압축률을 보이는 파일의 쓰기 수행 시간 비교
Fig. 4 Execution Time Comparison High Compression Rate File vs. Low Compression Rate File

그림 4은 파일의 압축률에 따른 파일 쓰기 수행 시간의 비교를 보여준다. 선택적 압축을 하지 않고, 사용자 수준의 압축파일을 압축 저장할 경우 중복 압축에 따른 불필요한 자원낭비와 쓰기 속도를 저하시킬 것이다. 파일 압축 시 사용자 수준의 압축 파일을 판단하기 위하여 파일의 일정 부분을 압축한 뒤 압축률을 검사한다. 압축률의 임계값을 설정하여 파일이 임계값 보다 낮은 압축률을 보일 경우 사용자 수준의 압축 파일로 판단하고 파일의 나머지 부분은 압축하지 않는다. 임계값보다 압축률이 높을 경우 파일의 나머지 부분도 압축을 수행하고 압축된 파일을 디스크에 쓰기한다.

파일 압축은 **gzip**포맷의 **zlib** 라이브러리를 사용하여 압축한다. **zlib** 압축 라이브러리는 거의 아무런 제한 없이 자유로이 사용할 수 있는 라이선스이며 메모리상에서의 압축 및 압축해제 함수들과 압축 해제된 데이터의 무결성 검사기능을 제공한다. 그리고 **zlib**는 안정성이 입증되어 압축한 다음 정상적인 압축 풀기를 보장하고, 같은 방식으로 다시 압축을 시도했을 때 더 이상 압축되지 않는다 [6]. 컴퓨터 시스템에서 아무런 성능저하 없이 공간을 절약할 방법을 찾기란 어렵다. 그러나 파일 시스템에서 가장 중요하게 여겨지는 안정성을 유지한 채 작은 비용으로 공간 활용을 개선할 수 있다면 저장 장치를 아주 효과적으로 사용할 수 있다. 리눅스 운영 체제에서 블록 디바이스 드라이버와 모듈 프로그램으로 구현한다 [8].

그림 5. 커널 모듈 프로그램 구조
Fig. 5 Kernel Module Program Structure

그림 5는 작성된 커널 모듈 프로그램의 구조이다. `register_blkdev`를 통하여 블록디바이스드라이버를 등록하며 실제적인 처리는 `ioctl()`함수를 통하여 이루어진다. 리눅스 운영체제에서 블록 디바이스 드라이버는 `ioctl()` 함수를 사용하여 읽기, 쓰기 처리가 가능하다. 블록디바이스에서 `ioctl()`함수의 명령(cmd)은 통일이 되어있으며 아래의 표 1과 같다 [1].

표 1. 블록 디바이스 드라이버의 CMD
Table. 1 CMD of the Block Device Driver

CMD	역할
BLKGETSIZE	섹터 수로 표현되는 블록 디바이스의 크기를 응용 프로그램에 전달
BLKFLSBUF	내부에 저장된 버퍼의 내용을 실제 블록 디바이스에 모두 써넣기를 요구
BLKRAGET	응용 프로그램에서 블록 디바이스에 설정되어있는 읽기값을 미리 얻어올 때 사용
BLKRASET	블록 디바이스의 미리 읽기 값을 설정
BLKRRPART	블록디바이스의 파티션 테이블을 다시 읽기를 요청
HDIO_GETGEO	블록 디바이스의 특성을 디스크 구조에 대한 형태로 응용 프로그램에 제공

`ioctl()`함수에서 `BLKFLSBUF`를 이용하여 내부에 저장된 버퍼의 내용을 블록디바이스 드라이버에 써넣으며 선택적 압축 알고리즘은 `BLKFLSBUF`에 구현한다. 모듈 프로그램은 `init_module()`과 `cleanup_module()`을 구현하여 커널에 적재, 제거시킨다.

그림 6은 커널 모듈 프로그램에서 `init_module()` 소스 코드이다. `init_module()`은 커널 콘솔에서 `insmod` 명령어를 이용하여 모듈 프로그램을 적재시킬 때 호출된다. `register_blkdev()`를 이용하여 모듈 프로그램에서 블록 디바이스 드라이버를 등록하며, 디바이스 드라이버의 Major Number와 디바이스 드라이버의 이름, 디바이스 드라이버에서 사용할 파일 오퍼레이션을 등록한다. 커널과 블록 디바이스는 `blk_queue_make_request()` 함수를 사용하여 실제적인 입출력 통로를 등록하며, 본 논문에서는 블록 디바이스의 크기를 4Mbyte로 지정한 후 실험 하였다 [1].

```
int init_module(void)
{
    int lp;

    register_blkdev(MAJOR_NR, VRD_DEV_NAME, &vrd_fops);

    blk_queue_make_request(BLK_DEFAULT_QUEUE(MAJOR_NR), &vrd_make_request);

    read_ahead[MAJOR_NR] = VRD_AHEAD;

    for( lp = 0; lp < VRD_MAX_DEVICES; lp++ )
        vrd_size[lp] = VRD_SIZE_KB;
    blk_size[MAJOR_NR] = vrd_size;

    vdisk[0] = vmalloc( VRD_SIZE );
    vdisk[1] = vmalloc( VRD_SIZE );

    for( lp = 0; lp < VRD_MAX_DEVICES; lp++ )
    {
        register_disk( NULL,
                        MKDEV(MAJOR_NR, lp),
                        1,
                        &vrd_fops,
                        VRD_SECTOR_TOTAL);
    }

    return 0;
}
```

그림 6. `init_module()`
Fig. 6 `init_module()`

```
void cleanup_module(void)
{
    int lp;

    unregister_blkdev( MAJOR_NR, VRD_DEV_NAME );

    vfree( vdisk[0] );
    vfree( vdisk[1] );

    read_ahead [MAJOR_NR] = 0;
    blksize_size[MAJOR_NR] = NULL;
    blk_size [MAJOR_NR] = NULL;
}
```

그림 7. `cleanup_module()`
Fig. 7 `cleanup_module()`

위 그림 7의 `cleanup_module()`은 리눅스의 콘솔에서 `rmmod` 명령어로 모듈 프로그램을 커널에서 제거할 때 호출 된다. `unregister_blkdev()` 함수에 사용한 블록 디바이스 드라이버의 Major Number와 디바이스 드라이버의 이름을 지정하여 블록 디바이스 드라이버를 `umount` 시키며 `vfree()` 함수를 사용하여 모듈 프로그램에서 사용한 메모리 할당을 해제한다.

IV. 실험 및 평가

본 장에서는 구현된 설계 내용을 바탕으로 실험한 내용의 성능 평가를 분석한다. 구현은 리눅스 커널 2.4.20-8 버전을 사용하는 RedHat 9.0 운영체제를 사용하였고, Core 2 duo 2.13GHz CPU와 2GB의 메모리를 탑재하고 있다. 실험은 EXT3 파일 시스템의 기본적으로 사용하는 Ordered mode를 사용하였으며, zlib 1.2.5 버전을 사용하고 gcc를 이용하여 프로그래밍과 컴파일을 하였다 [2]. 실험은 기존의 EXT3 파일 시스템과 선택적 압축 알고리즘이 적용된 EXT3 파일 시스템의 성능 비교를 위해 멀티미디어 파일을 제외 후 무작위로 선택된 50개의 파일을 임의로 적용하여 실험을 하였다. 성능 비교를 위해 기존의 EXT3 파일 시스템과 선택적 압축 알고리즘이 포함된 EXT3 파일 시스템의 전체 파일 크기를 비교 분석하였다. 실험 파일의 앞부분 10240byte만큼 압축을 실행한 뒤 압축된 파일의 크기가 임계값으로 설정한 6800byte보다 작을 경우 선택적 압축 저장기법을 통하여 쓰기 수행이 되도록 구현되어 있다. 원본 파일 크기의 약 2/3 이하로 압축률을 보일 경우이다. 기존 EXT3 파일 시스템에서의 50개 파일의 총 크기는 24392704byte이며, 개선된 EXT3 파일 시스템에서의 총 파일 크기는 14342756byte로 약 41%의 공간적 효율이 발생하였다. 전체 50개 파일 중 9개의 파일이 압축률이 낮아 원본저장이 되었고 41개의 파일이 선택적 압축 알고리즘을 통하여 압축 저장되었다. 압축 수행시간은 미비하여 공간적 효율에 비해 무시해도 될 정도이다. 아래의 그림 8은 실험한 결과를 그래프로 보여준다.

그림 8. 파일 시스템별 총 파일 크기 비교

Fig. 8 Total File Size Comparison for File System

많은 실험 데이터의 결과를 바탕으로 일반 데이터 파일의 각 파일 포맷별 헤더 부분 압축 효율이 다르기 때문에 선택적 압축 알고리즘에서 앞부분 압축 시킬 크기와 임계값의 크기 조절로 보다 효율적인 임계값의 크기를 구할 수 있다.

V. 결 론

본 논문에서 EXT3 파일 시스템 기능 개선 연구는 기존의 EXT3 파일 시스템의 ordered mode에 압축 저장 기법 적용을 제안한다. 기존 EXT3의 파일 쓰기 시 저장 장치의 효율적인 공간 활용과 쓰기속도를 향상시키기 위해 선택적으로 압축 저장 기법을 적용 시킨다. 파일 시스템의 안정성에 문제가 생기면 파일 시스템의 동작이 중단되어 수정 중이던 데이터가 손상되거나 복구가 불가능하게 될 수도 있다. 본 논문에서 제안하는 EXT3 파일 시스템 기능 개선 연구는 기존 EXT3 파일 시스템의 ordered mode 저널링을 수행하기 때문에 안정성을 그대로 유지할 수 있다. 본 논문에서 제안하는 개선된 EXT3 파일 시스템에서 압축률이 높은 파일은 추가적인 파일 압축 과정을 수행하지만 파일 쓰기속도 향상과 디스크 공간을 효율적으로 사용할 수 있을 것이다. 하지만 사용자 수준의 압축 파일등과 같은 압축률이 낮은 파일의 경우에도 파일의 일정 부분이 압축 작업이 수행되기 때문에 성능상 단점이 발생 할 것이다.

참고문헌

- [1] 이현철, “효율적인 메모리 사용을 위한 LZCode 기반의 압축 기법”, 아주대학교 석사학위논문, 2012.
- [2] 김용관(Yongkwan Kim) 위영철(Youngcheul Wee), “휴대장치를 위한 고속복원의 프로그램 코드 압축 기법”, 정보과학회논문지 : 소프트웨어 및 응용, Vol.37 No.11, 2010.
- [3] 전창규(Chang Kyu Jeon) 류경식(Kyeong Seek Lew) 김용득(Yong Deak Kim), “임베디드 시스템에서 실행 가능 압축 기법을 이용한 프로그램 초기 실행 속도 향상”, 電子工學會論文誌-CI, Vol.49 No.1, 2012.

- [4] 이성현(Seong-Heon Lee) 장승주(Seung-Ju Jang),
“데이터 압축을 통한 효율적인 저장 공간 사용을
보장하는 저널링 파일 시스템 설계”, 한국정보과학
회 학술발표논문집, Vol.38 No.2A, 2011.
- [5] 조범석(Beom-Seok Joh) 김영로(Young-Ro Kim),
“가변적 준무손실 압축 알고리즘 = Variable Near-
Lossless Compression Algorithm”, 대한전자공학회
학술대회 논문집, Vol.2010 No.6, 2010.
- [6] 조미남(Mi-Nam Cho) 지유강(Yoo-Kang Ji), “임베
디드시스템을 위한 혼용텍스트 파일의 개선된
LZW 압축 알고리즘 구현”, 한국콘텐츠학회논문
지, Vol.10 No.12, 2010.
- [7] The Linux Cross Reference, <http://lxr.linux.no/>
- [8] Linux kernel source code, <http://www.kernel.org/>
- [9] Daniel Robbins, “고급 파일 시스템 개발자 가이드,
part7 & 8”
- [10] TAKAHASI HIROKAZU, “리눅스 커널 2.6 구조와
원리”, 한빛미디어, 2007
- [11] Daniel P.Bovet, Marco Cesati, “Understanding the
LINUX KERNEL”, O'Relly, January 2001.
- [12] Korean Linux Documentation Project, <http://kldp.org/>

저자소개

장승주(Jang, Seung Ju)

1985년 부산대학교
계산통계학(전산학) 학사
1991년 부산대학교
계산통계학(전산학) 석사

1996년 부산대학교 컴퓨터공학 박사
1987년 ~ 1996년 한국전자통신연구원(ETRI) 시스템
SW연구실
1993년 ~ 1996년 부산대학교 시간강사
2000년 ~ 2002년 Univ. of Missouri at Kansas City,
visiting professor
1996년 ~ 현재 동의대학교 컴퓨터공학과 교수
※관심분야: 운영체제, 임베디드 운영체제, 분산
시스템, 시스템 보안, 스마트 폰 시스템 운영체제