|  |
| --- |
| LZ4 압축 기법을 활용한 디바이스 드라이버 압축 구현 |
|  |
| Implementation of Block Device Driver Compression using LZ4 |
|  |
| **요 약**  기존의 File system LZ4 압축 기법을 활용하여 Data compression을 통해 Compressed block device driver 를 구현한다. 본 구현을 통해 원본 데이터는 약 16% 정도 압축률을 보이며, 실행 시간의 경우 약 2배 정도 성능 차이를 보인다. | | |

**1. 서 론 (Introduction)**

Linux kernel에는 데이터를 압축하기 위한 다양한 compression 기법이 존재한다. 이러한 데이터 압축 기법은 performance와 data compression rate 간의 trade-off가 존재한다.

먼저, Data compression rate를 낮추면, performance가 증가하게 된다. 하지만 이를 통해 Device에 더 많은 데이터를 저장할 수 없게 된다. 반대로, Data compression rate를 높게 하면, performance가 저하 되지만 Device에 더 많은 데이터를 저장할 수 있게 된다.

이러한 trade-off 알아 보기 위해 본 프로젝트에서는 Linux kernel에서 지원 하는 압축 기법인 LZ4 compression 기법을 사용하여 Compressed block device driver를 구현하여 Compression rate에 따라 보여지는 성능 차이를 관찰하고자 한다.

본 구현을 통해 구현된 Compressed block device driver는 원본 데이터의 약 17~20% 정도의 Compression rate를 보이며, 실행 시간의 경우 약 2배 정도 성능 차이를 보였다.

**2. LZ4**

LZ4는 Yann Collet이 개발한 멀티코어를 지원하는 비손실 압축 알고리즘이다. 압축률은 기존 압축 알고리즘인 Deflate등에 비해 낮지만 속도가 최대 10배 이상으로 매우 빠르다. C로 구현이 되어있고 BSD라이센스를 따른다. 다양한 프로그래밍 언어로 포팅 되거나 바인딩 되어 있다. 현재 LZ4는 file system 압축 기법 중 하나로 Linux kernel에서 Data compression 기법으로 널리 사용 되고 있다. Linux kernel에서 지원하는 LZ4 API를 활용하여 Compressed block device driver를 구현하였다.

**3. 구 현 (Implementation)**

3-1 LZ4 API

본 프로젝트는 Linux file system 압축 기법인 LZ4 API를 활용하여 구현 되었다. 구현을 자세히 설명하기 전에, 사용된 주요 API 들에 대한 설명 후, 구현에 대해 세부사항을 전달하고자 한다. 구현을 위해 LZ4\_compress\_default 함수와 LZ4\_decompress\_safe 함수가 사용 되었다. 먼저 Compression 하기 위한 LZ4\_compress\_default() 함수의 형태는 다음과 같다.

LZ4\_compress\_default(

Char \*src,

Char \*dest,

Int inputsize,

Int maxOutputSize,

Void \*wrkmem)

이 중 maxOutputSize와 wrkmem의 인자는 LZ4 함수 내부의 정의된 값으로 넣어 주어야 한다. 먼저, maxOutputSize는 원본 데이터가 압축이 잘 되지 않을 때, LZ4가 지원할 수 있는 최대 압축 크기를 나타낸다. 이를 활용하기 위해서 LZ4\_compressBound() 함수를 사용한다. 이는 input 데이터에 대한 최대 크기를 정해주는 함수 이다 이러한 함수를 활용하여 maxOutputSize를 구하기 위한 예시는 다음과 같다.

Ex)

int maxOutputSize = LZ4\_compressBound(inputsize)

다음은 wrkmem에 관해서이다. 이는 LZ4를 압축하기 위해 필요한 kernel에서 사용될 memory를 나타낸다. kernel에서의 정의는 다음과 같다.

*@wrkmem: address of the working memory.*

*\* This requires 'workmem' of LZ4\_MEM\_COMPRESS.*

그러므로, wkmem을 사용하기 위해서는 위의 언급된 LZ4\_MEM\_COMPRESS 상수 값만큼의 메모리가 요구된다. wkmem 메모리 할당이 끝난 후, LZ4\_compress\_default()의 세팅이 완료된다. 이 함수가 성공적으로 압축을 완료한다면, compressed size를 return하고, 그렇지 않다면 negative value를 return 한다.

다음은 decompression 함수인 LZ4\_decompress\_safe 함수는 compression함수에 비해 간단하다.

LZ4\_decompress\_safe(

Char \*src,

Char \*dest,

int compressedSize

int maxDecompressedSize)

이 중 compressedSize는 원본데이터에 대한 압축된 크기를 나타내고, maxDecompressedSize는 decompression에 했을 때, 최대로 decompression 될 수 있는 크기를 나타낸다. 이를 위해 compressedSize는 저장해 두어야 한다. 이는 구현에서 자세히 설명할 것이다. 또한 maxDecompressedSize는 본 프로젝트에서는 page 크기(4096bytes)로 지정 되어 있다.

3-2 Implementation of Compressed block device driver

위의 언급된 LZ4 함수들을 이용하여 Compressed block device driver는 간단하게 구현되었다. 구현된 코드는 다음과 같다.

sbull\_transfer()

{

If(write)

{

c\_offset = offset / Block\_size(4096)

c\_size = LZ4\_compress\_default()

arr[c\_offset] = c\_size;

}

else

{

d\_offset = offset / Block\_size(4096)

size = array[d\_offset];

d\_size = LZ4\_decompress\_safe(size)

if(arr[d\_offset] != -1)

{

Increment log\_counts

}

}

}

이 구현은 블록 크기 만큼 request들을 압축하기 위해 sbull\_transfer 함수에 구현 되었다.

위의 코드의 핵심은 compress\_size를 관리하기 위한 integer array이다. Request(Write)가 들어올 때, device에 offset에서 compressedSize 만큼 저장하게 된다. Offset과 size 정보를 관리하기 위해 array가 필요한 것이다. 이러한 array 정보는 decompression (read) 할 때 사용되고 있다. (3-1에서 언급된 CompressedSize 관리)

**4. 실 험 (Evaluation)**

4-1 Evaluation Setting

System configuration

OS : ubuntu 16.04

Kernel version : 4.12

**실행 방법**

1. 20182201.zip -> 압축 해제 -> sbull.tar.gz 해제

**sbull 폴더**

{ sbull.c sbull.h, **project**폴더, .. , Makefile }

**project 폴더**

bench1.c bench2.c Makefile (sample1, sample2는 추가 되어야함)

1. cd sbull 🡪 make

sudo modprobe lz4 (필수)

./sbull\_load

lsblk 🡪 Check 1GB sbulla device

cd sbull/project 🡪 make 🡪 bench1, bench2

./bench1 or ./bench2

\*\* 주의 사항 \*\*

1. 반드시 project 내부의 bench1.c bench2.c 코드를 사용해야함 (기존 코드 수정했기 때문)
2. bench 실행할 때마다 pagecache를 비워야 함

echo 3 > /proc/sys/vm/drop\_caches

1. Sudo 권한으로 모든 것을 실행 해야 함

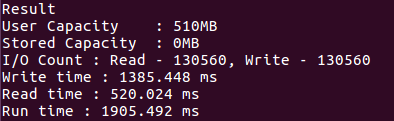


Figure 1 – Not compression(bench1)

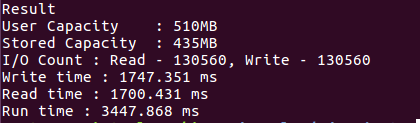


Figure 2 – LZ4 compression(bench1)

위의 실험 결과들은 LZ4 compression 기법을 활용하여 구현된 compressed block device driver의 bench1 결과이다. 먼저 Fig 1은 compression 기법을 적용하지 않은 경우의 결과이다. Compression 되지 않기 Storage Capacity는 0이며, Run time은 1905ms 정도 소요되었다. 반면 Fig 2는 LZ4 compression 기법을 적용했을 때 결과이다. 결과에서 볼 수 있듯이, 원본데이터 510MB에서 435MB (16%)로 압축된 것을 확인 할 수 있다. 하지만 이러한 압축으로 인해 Fig 1 과 비교했을 때, 약 2배 정도의 성능 차이를 보이고 있는 것을 확인할 수 있다.

다음 오른쪽 그림(Fig 3, Fig4)들은 bench2에 대한 실험 결과이다. bench2에 대한 실험 결과도 bench1에 대한 실험 결과와 거의 비슷하다. 하지만 bench2와 bench1의 실험 결과의 차이점은 write에 대한 성능 차이가 보이고 있는 점이다. 이는 bench1과 bench2의 workload 특성에 따라 달라 지는 결과를 보이는 것이다. bench1의 경우는 파일의 처음부터 끝까지 sequential하게 읽지만 bench2의 경우는 파일 offset을 random하게 읽어 write를 한다. 이때 bench1의 경우 sequential 특성 때문에 page caching 되지 않으므로 device에서 파일을 계속 읽어야 하기 때문에 write에 대한 성능이 저하된다. 반면 bench2의 경우 random하게 파일을 읽기 때문에 해당 다음 request가 page caching 되 있는 경우가 발생하게 된다. 따라서 device에서 파일을 읽지 않아도 되기 때문에 write 성능이 bench1보다는 증가하는 것이다.

**5. 결 론 (Conclusion)**

Linux kernel에는 Data compression을 위한 다양한 기법을 지원하고 있다. 이 중 LZ4는 Data compression 에 대한 다양한 API를 제공하며 Compressed block

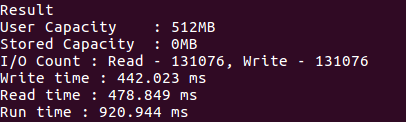


Figure 3 – Not compression(bench2)

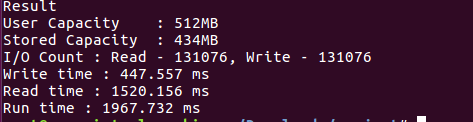


Figure 4 – LZ4 compression(bench2)

device driver를 구현하는 데 유용하게 활용되었다. 이를 토대로 실험 결과 16%의 데이터 압축률을 보였다. 하지만 그에 따른 trade-off로 약 2배정도의 성능 저하가 나타났다. 이러한 영향은 Data compression 과 performance의 trade-off 분명히 존재한다는 것을 확인하게 해준다. 따라서 시스템 특성에 따라 Data compression 기법을 고려해야 한다.