SANfs: 리눅스 클러스터를 위한 확장성있는 공유파일시스템

황주영[0], 안철우, 박규호
한국과학기술원 전자전산학과 전기전자전공
(jyhwang, cwahn, kpark)@core.kaist.ac.kr

SANfs: A Scalable Shared File System for Linux Clusters

Joo Young Hwang[0], Chul Woo Ahn, Kyu-Ho Park
EECS Dept. Korea Advanced Institute of Science and Technology

요 약

본 논문에서는 Storage Area Network기반의 대규모용량의 저장장치에 적합한 확장성있는 공유파일시스템을 제안하고, Linux상에서의 구현을 보인다. 락의 단위는 블록이 아닌 파일단위로 하므로써 락오버헤드를 줄이고, callback방식의 파일 락을 사용한다. 파일데이터일관성 및 디렉토리캐쉬 일관성유지를 위한 Linux상에서의 구현방법을 보인다. 메타서버의 free block관리부담을 줄이기 위해서 분산 free block관리법을 사용한다. 또한 Inode와 data를 분리 저장함으로써 성능을 최적화한다.

1. 서 론

최근의 Storage Area Network(SAN) 기술의 발전은 저장장치 시스템에 획기적인 발전으로 가져왔다. SAN은 고속의 스위칭 네트웍으로서, 1Gbps의 속도를 지원하며, 추후 4Gbps까지 지원할 수 있다. 또한 수백개의 대용량 디스크를 연결할 수 있기 때문에 peta급의 저장공간을 구현할 수 있게 되었다.
SAN에 기반한 저장장치를 효율적으로 관리하기 위해서는 데이터를 여러 호스트들이 데이터일관성을 유지하면서 공유할수 있도록 하는 파일시스템이 절실히 필요하다.
이 논문에서는 확장성이 우수한 공유파일시스템을 디자인하고 Linux상에서 구현하였다.

2. 관련 연구

GFS[1][5]는 Linux 상에서 구현된 공유파일시스템으로서, disk lock(dlock)을 사용하여, 데이터 일관성을 지원한다. 기존의 FFS 구조에서는 direct block, indirect block, double indirect block, triple indirect block등으로 block의 indexing 계층구조가 구성된 반면 GFS는 모든 데이터 블록이 indirect block으로 구성된다. Inode는 SAN디스크에 저장되며, 하나의 disk block에 매핑되어 있다. SANTopia[6]는 semiflat inode 구조 및 metadata journaling기법을 제안하였다. Storage Tank[4]는 centralized locking server를 사용하여 file consistency를 지원한다. Centralized locking server는 single point of bottleneck이 될 가능성이 있으나, 기존의 분산 파일시스템에서의 data server와는 다르게 consistency control만을 담당하므로 bottleneck이 되지 않는다. 또한 GFS에서 지원할 수 없는 다양한 파일오픈모드를 지원한다. SAN환경에서는 모든 호스트가 모든 디바이스를 로컬 디바이스처럼 액세스할 수 있기 때문에 악의적으로 디스크데이터를 변경할 수 있다. 따라서 디스크데이터에 대한 접근권한을 관리하는 것이 필요하며, storage tank는 이러한 접근권한관리서비스 방법을 제안한다.

3. SANfs 공유파일시스템

SANfs파일시스템은 대규모 저장장치를 겨냥하여 탁월한 확장성을 갖도록 설계하였으며, Linux상에서 개발되었다. 파일시스템의 각 파트의 디자인 및 구현에 관하여 기술한다.

3.1 파일 데이터 일관성 유지

서로 다른 호스트간의 동일한 파일을 공유하면서 쓰는 경우는 거의 일어나지 않는다.[2] block단위의 consistency를 지원하는 것은 복잡하며, 불필요하다. 파일은 여러 호스트에 의해서 읽기공유는 허용되고, 쓰기공유는 허용되지 않는다. 또한 파일의 쓰기권한을 가진 호스트는 그 파일의 모든 블록들에 대해서 쓰기권한을 갖게 된다.
중앙메타서버가 파일일관성유지를 위해서 파일 락을 관리한다. 각 호스트는 파일을 열 때 메타서버로부터 해당 락을 요청한다. 락종류는 읽기전용락, 읽기/쓰기락 두 가지가 있다. 호스트는 메타서버로부터 락을 파일을 닫은 뒤에도 계속 가지고 있을 수 있다, 이것은 최근에 액세스한 파일을 다시 액세스하게 되는 가능성이 높다는 것을 이용하여 메타서버와의 통신회수를 줄이는 방법이다. 메타서버가 락을 릴리스하도록 요청이 오면, 파일의 수정된 블록들을 모두 디스크에 쓴 다음 락을 릴리스한다. 이것은 Frangipani파일시스템의 sticky lock과 유사하다.[3]
메타서버는 user-level program으로 구현되었다. 파일시스템 호스트와는 하나의 TCP socket으로 통신한다. 단일 호스트의 여러 프로세스들이 하나의 socket을 공유하면서 서로 다른 세션을 통해서 통신한다. 각 프로세스는 메타서버와 세션을 열고 요청패킷을 보낸다. 응답패킷이 오면 세션관리매니저가 해당프로세스에게 전달한다. 패킷의 데이터 구조는 따음과 같다. I_ino는 inode number이며, lock_type은 락의 종류를 지정한다. 각 프로세스는 session id를 할당받고나서, 패킷의 session_id를 실어서 요청을 보낸 다음, sleep한다. 세션매니저는 메타서버로부터 오는 모든 패킷을 받아서 패킷내의 session_id를 참조하여 해당 프로세스를 wakeup한다.

```
struct request_packet {
   int i_ino; // inode number
   int lock_type; // read-lock or read-write-lock
   int session_id;
};

struct reply_packet {
   int i_ino; // inode number
   int lock_type; // read-lock or read-write-lock
   int session_id;
   struct inode inode_data; // inode data
};
```

파일락을 릴리스할 때 해당 파일의 모든 수정된 블록들을 디스크에 쓰기 위해서는 버퍼캐쉬내의 파일의 블록들을 linked list로 관리한다. 블록이 버퍼캐쉬에 올라올 때마다 해당 파일의 I_buffers 리스트에 연결되고, 캐쉬에서 대치될 때 리스트에서 제거된다.
기존의 Linux VFS에서는 위와 같이 파일의 블록들을 알수 있는 방법이 없다. 블록이 버퍼캐쉬에 없으면 getblk(dev, block, size)를 호출하여 dev로부터 block을 읽어오고, 이것을 위한 buffer_head를 초기화하여 insert_into_queues(struct buffer_head* bh)를 호출하므로써 buffer_head가 LRU list에 들어가게 된다. 이 과정에서 블록이 어떤 파일에 소속된 것인지를 관리하기 위해서 SFSgetblk함수가 새로이 추가되었다. Struct buffer_head구조체에 추가된 b_inode는 블록이 소속된 inode로의 pointer이고, b_ibnext, b_ibprev는 inode의 I_buffers리스트에 연결하기 위한 커넥터이다.

```
struct buffer_head * SFSgetblk(struct inode* inode, kdev_t
dev, int block, int size)
{
  ...
repeat:
   bh = get_hash_table(dev, block, size);
   if( bh ) {
     ...
     return bh;
   }
   isize = BUFSIZE_INDEX(size);
get_free:
   bh = free_list[isize];
   ...
   insert_into_queues(bh);

   bh->b_inode = inode;
   blist_add_buffer(&bh->b_inode->I_buffers, bh);
   return bh;
}
```

파일이 제거될 때는 메타서버가 제거되는 파일을 모든 호스트에게 broadcast하고, 각 호스트는 버퍼캐쉬에 제거될 파일의 블록들을 모두 버퍼 LRU리스트에서 제거한다. (remove_from_hash_queue) 이 때 blist_del_buffer를 호출하여 inode I_buffers 리스트에서 buffer_head를 제거한다.

3.2 Nested 락 호출 관리

파일시스템 각 함수들마다 필요한 inode lock을 함수 도입부에서 lock하고, 함수종료부에서 unlock한다. 한 함수에서 다른 함수를 호출하게 되는 경우 이미 lock이 된 inode에 대해서 다시 lock을 하게 되는 데, 이 때 메타서버로 락요청을 보내지 않고 lock_depth를 1 증가시키기만 한다. Lock_depth는 또한 다른 프로세스가 inode를 사용할 때에도 증가된다. 아무 프로세스도 inode를 사용하지 않을 때는 lock_depth는 0이 된다. 예를 들어서, 다음과 같이 func2내에서 func1을 call하면, inode는 이미 메타서버로부터 lock을 가져온 상태이므로 lock_depth만을 1증가시킨다. 또한 unlock에서는 lock_depth를 1감소시킨다.

```
Void func1 ( inode ) {
   Lock(inode);
   .....
   Unlock(inode);
}

void func2(inode){
   Lock(inode);
   .....
   func1(inode);
   ...
   Unlock(inode);
}
```

3.3 디렉토리캐쉬 일관성 유지

Linux에서는 directory cache를 사용하여 directory lookup의 결과를 캐쉬하므로써 filename-to-inode translation의 성능을 향상시킨다. 공유파일시스템에서는 서로다른 호스트의 directory cache의 일관성을 유지하는 것이 필요하다. Directory lookup함수내에 directory를 lock하도록 하면 공유파일시스템이 아닌 다른 파일시스템의 성능을 저하시키게 되므로 바람직하지 않다. 한 호스트에서 파일이 지워진 경우에는 메타서버로 delete를 알리고, 다시 메타서버가 모든 호스트에게로 delete를 broadcast한다. Delete message를 받은 호스트는 각자의 directory cache에서 해당 파일을 제거한다. 호스트가 메타서버로부터 delete message를 받으면 deleted 플래그를 1로 세팅하고, I_nlink를 0로 한다음 d_delete를 호출한다. Inode를 inode hash에서 제거하고, directory entry를 directory cache에서 제거하는 것은 dentry_iput()에서 일어난다. Iput함수에서는 해당 inode를 delete하기 위해서 파일시스템 specific delete함수를 호출한다. SANfs에서는 deleted flag가 setting된 inode에 대해서는 truncate와 free inode작업이 이루어질 필요가 없고 단지 I_buffer리스트에 연결된 모든 블록들을 LRU에서 제거한다.

```
static void fdfs_delete_inode(struct inode *inode)
{
    if( inode->u.fdfs_i.deleted ) {
      fdfs_remove_ibuffers(inode);
    } else {
      inode->i_size = 0;
      fdfs_truncate(inode);
      fdfs_free_inode(inode);
    }
}
```

3.4 분산 Free block 관리

파일시스템의 free block은 메타서버에서 집중관리하며, 호스트는 free block이 필요할 때마다 메타서버에서 가져온다. 메타서버의 부담 및 통신량을 줄이기 위해서 메타서버에서 한번에 2,048개의 block chunk를 가져온 다음 local process들의 free block요청을 서비스한다. Free block들은 unmount시에 메타서버로 되돌린다.

3.5 inode와 data의 분리

inode정보는 메타서버의 메모리 및 로컬디스크에서 관리하도

록 하고, 사용자 데이터는 SAN디스크에 저장하도록 하므로써 SAN을 고성능 데이터입출력에 최적화시킨다. Inode정보가 SAN 디스크에 같이 저장되어있으면, small read/write가 빈번히 일어나게 되므로 SAN의 대역폭을 효율적으로 사용할 수 없게 된다. Inode와 data를 분리시켜 각각의 액세스 방식에 맞게 최적화시킬 수 있다. Inode정보는 파일락의 응답 패킷에 piggyback하므로써 통신오버헤드를 줄인다.

4. 실험 결과

구현된 시스템은 4대의 9GB Seagate Fibre-channel SCSI disk와 4대의 PC로 구성되었다. 각 PC는 256MB의 메모리와, 2개의 450MHz Pentium CPU를 가지고 있다. OS는 Linux kernel 2.2.12를 사용하였다.
PC 한 대는 메타서버와 파일시스템 클라이언트 모두로 동작하도록 구성하고, 나머지 3대는 파일시스템 클라이언트로만 동작하도록 하였다.
SANfs 파일시스템의 성능을 측정하기 위해서 각 파일시스템 클라이언트는 4개의 서로 다른 디스크의 파일시스템을 마운트하여 랜덤하게 액세스하도록 하였다. 디스크 한대의 성능은 15MB/sec이다. 그림 1에 보인 바와 같이 호스트의 개수가 증가함에 따라 전체 시스템의 성능은 거의 선형적으로 증가한다. 이것은 메타서버가 병목이 일어나지 않음을 보여준다. 성능이 디스크의 최고성능에 미치지 못하는 것은 블록의 사이즈가 1K이기 때문이다. 성능을 높이기 위해서는 4K나 8K로 블록사이즈를 증가시켜야 한다.

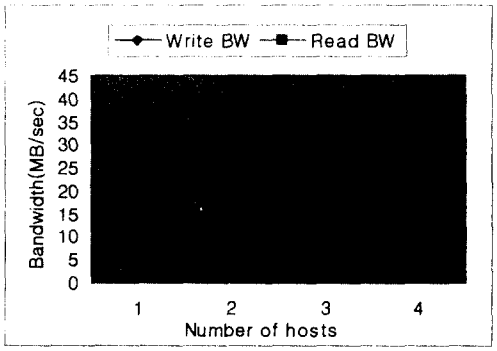

그림 1 Scalable performance of SANfs shared file system

5. 결론 및 추후과제

본 논문에서는 SAN기반의 저장시스템을 위한 확장성있는 공유파일시스템을 제안하고, Linux상에서의 구현하였다. 대규모 저장장치를 효율적으로 관리하기 위해서 오버헤드가 적은 file 단위의 locking방식을 사용하였다. 파일단위로 데이터의 일관성유지를 위해서 기존의 Linux VFS의 buffer cache 관리를 수정하였다. Callback방식의 파일 락을 사용하여 메타서버와의 통신량을 줄인다. 또한 directory cache일관성을 유지하는 프로토콜을 제안하였다. Free block 관리는 메타서버에서 집중관리하며, 각 호스트가 메타서버에서 block chunk를 가져와서 local 프로세스에게 서비스하도록 하므로써 메타서버의 관리부담을 줄인다. Inode와 데이터를 분리하여 저장하므로써 각각의 액세스 방식에 따라서 최적화시킬 수 있다.
앞으로의 연구과제로는 메타서버의 병렬화, extent-based block indexing방식구현, SAN상에서의 disk scheduling최적화 등이 있다. 블록사이즈를 1K로 고정시키지 않고 가변적으로 조정하도록 하여 성능을 높여야 한다. 또한 편리한 운용을 위한 관리소프트웨어를 개발할 예정이다.

참고문헌

[1] K.W. Preslan,et. al. " A 64-bit, shared disk file system for Linux" , Mass Storage Systems, 1999. 16th IEEE Symposium on ,1999 pages 22-41
[2] J.Ousterhout, H. Dacosta, D.Harrision, J.Kunze, M.Kupfer, and J. Thompson. " A trace-driven analysis of the unix 4.2 bsd file system" , In Proceesings of the 10th ACM symposium on Operating system Principles, Orcas Island, WA., December 1985.
[3] C. A. Thekkath, T. Mann, and E. K. Lee. " Frangipani: A Scalable Distributed File System." In ACM Symposium on Operating System Principles, 1997.
[4] Randal C. Burns, Robert M. Rees, Darrell D. E. Long, "Semi-Preemptible Locks for a Distributed File System", Performance, Computing, and Communications Conference, 2000. IPCCC '00. Conference Proceeding of the IEEE International, 2000 pages 397-404
[5] Steven R. Soltis, Thomas M. Ruwart, Matthew T. O'Keefe, " The Global File System" , In the Fifth NASA Goddard Conference on Mass Storage Systems and Technologies, volume 2, pages 319—342, College Park, Maryland, March 1996.
[6] Yong Kyu Lee; Shin Woo Kim; Gyoung Bae Kim; Bum Joo Shin, " Metadata management of the SANtopia file system" , Parallel and Distributed Systems, 2001. ICPADS 2001. Proceedings. Eighth International Conference on , 2001 Page(s): 492 -499