snapPVFS: Snapshot-able Parallel Virtual File System

Kwangho Cha[†‡], Jin-Soo Kim[†] and Seungryoul Maeng[†]
[†]Computer Science Department, Korea Advanced Institute of Science and Technology, Daejeon, KOREA
[‡]Supercomputing Center, Korea Institute of Science and Technology Information, Daejeon, KOREA
{khocha@camars, jinsoo@cs, maeng@camars}.kaist.ac.kr

Abstract

In this paper, we propose a modified parallel virtual file system that provides snapshot functionality. Because typical file systems are exposed to various failures, taking a snapshot is a good way to enhance the reliability of file systems. The PVFS, which is one of the famous parallel file systems deployed in cluster systems, is vulnerable to system failures or users' mistakes; however, there is a scarcity of research on snapshots or online backup for the PVFS. Because a PVFS consists of multiple servers on a network, snapshots should be generated properly in each server in the system. Furthermore, before snapshots are generated, the status of each PVFS server must be checked to guarantee sound operation. To demonstrate our approach, we implemented two prototypes of a snapshot-able PVFS (snapPVFS). The performance measurements indicate that an administrator can take snapshots of an entire parallel file system and properly access any previous versions of files or directories in the future without serious performance degradation.

1 Introduction

The parallel virtual file system (PVFS) is an open source-based parallel file system that utilizes multiple data server machines connected to a system or storage area network to provide linearly scaling performance. Furthermore, the PVFS enables parallel file systems to be easily constructed and operated with general purpose components. The independence and openness of the PVFS has made it the prevalent file system in cluster computing systems[1].

In addition to performance, file systems must satisfy reliability requirements. According to a recent study on the reliability of large-scale systems, storage is one of the main reasons of system failure [2]. This situation, which seems to have worsened because of the expected growth in storage systems, is the reason we insist that file systems in large-scale systems should be reliable.

In the case of large distributed memory computer systems, especially supercomputers, parallel file systems are generally used for job scratch space, which is mainly used to store large volumes of transient data [3]. Unlike user home directories, which are usually set to store programs, libraries and documents, the data in the job scratch space are seldom backed up [4]. Backing up generally takes a long time, even longer than a day [5]. Hence, modern backup systems are unsuitable for the job scratch space.

Although the job scratch space contains transient data, problems in the data can degrade the availability of the entire system. For example, if an intermediate file in the job scratch space is broken or erased, it may be necessary to resubmit a job to regenerate the lost file. The method of recovering transient data helps prevent the squandering of computational resources [3].

However, because the current version of PVFS has no online backup or snapshot support, we implemented a prototype snapshot-able PVFS (snapPVFS) by modifying a PVFS and the Linux-based versioning file system ext3cow. Our results confirm that snapshots of a file system can be taken and retrieved properly. The preliminary performance measurements also show that snapPVFS can generate snapshots without seriously degrading performance.

This paper is organized as follows: We summarize the background of our research in section 2. Section 3 presents the overall architecture of our proposed scheme, snapPVFS. The performance measurements are described in section 4. Finally, we present our conclusions in section 5.

2 Related work

2.1 Parallel virtual file system

One of the well-known parallel file systems is the PVFS [1]. The PVFS can provide high-performance I/O capabilities by striping blocks of data files across multiple disks on multiple storage nodes because of the possibility of reading or writing these blocks simultaneously. There may be any number of servers and each server may provide either metadata, file data or both. Metadata includes attributes such as

1521-9097/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPADS.2008.51

timestamps, permissions, and file system specific parameters; file data means the actual data stored in the system. Our approach is based on version 2.6 of the PVFS, which consists of five major components: a buffered message interface (BMI), flows, a trove, a job interface, and state machines [1][6].

2.2 Versioning file system

The major goal of data versioning is to protect data in storage and file systems from system failures or unexpected misuse. Versioning techniques include a file system snapshot and individual file versioning. A snapshot in this context means a read-only, unchangeable, and logical image of a collection of data at a certain point of time. Snapshots of a file system are generally used for backup, archiving and data mining. A snapshot can be implemented at the logical file system level and the disk storage level. A more detailed explanation of the snapshot is given in the next section.

File versioning creates a new logical version on every disk write or on every open or close session. In this study, we assume that the versioning file system is a file system with versioning features. The various designs of versioning file systems have different functions and implementation environments. The elephant file system introduced four retention policies. Depending on which policy is adopted, file systems can have a different undo period and important file versions. To reflect the retention policies and maintain multiple inodes, the elephant file system uses three metadata structures: an inode file, an inode log, and an imap [7].

Wayback is a user-level versioning file system for Linux. The developers of Wayback focused on easy implementation and portability of their new file system. When Wayback traps a system call for files with the help of a file system in user space, it writes an undo log. By applying the undo log in reverse order, Wayback can go backward in time [8].

The ext3cow file system, which is based on an ext3 file system, improves the functionality of the ext3 file system with versioning. It also provides a time-shifting interface that supports a real-time view of previous data. Users can generate a snapshot of the entire file system and retrieve the snapshot as a read-only image. The inodes of ext3cow were changed to support snapshot and copy-on-write functions by the addition of three fields: an inode epoch counter, a copy-on-write bitmap, and a field that points to the next inode in the version chain [9].

2.3 Snapshot

A snapshot generally refers to the ability to record the state of a storage or file system; it can be found in various storage-related components, such as file systems, volume managers, and storage arrays. Because a snapshot provides easy backup of a large volume of data and point-in-time copies of data, it facilitates recovery from system corruption and simplifies analysis of historical changes of data. Moreover, it can protect data from the mistakes of users, such as accidental deletion [10][11].

The copy-on-write function is the one of the prevailing ways of creating snapshots. For the initial snapshot, the copy-on-write function merely copies the metadata pertaining to the location of the original data. If there are any write requests for snapshotted data, the original data is moved to a predesignated space before it is overwritten. In this method, the creation of a snapshot is almost instantaneous and the snapshot space only holds the changed data. However, it has a double write penalty, which means that any change to the snapshotted data yields two write requests: one for moving the original data and one for changing the data [10].

The redirect-on-write function is similar to the copy-on-write function. Because new writes to snapshotted data are redirected to another space, the double write penalty of the copy-on-write function is eliminated. However, when a snapshot is deleted, the redirect-on-write function must perform more complicated routines than those in the copy-on-write function [10].

IBM's GPFS is a commercial high performance file system that is available on AIX and Linux. Because the GPFS has the features of high scalability, data stripping, concurrent file access, block-level locking, high availability and large block size, it has become a famous parallel file system and is used in many supercomputers[12].

In addition to being a backup solution, the GPFS can provide a snapshot of the file system. A user with root authority can create a snapshot of an entire file system by using the mmcrsnapshot command. In the case of restoring a system from a snapshot, the file system should be unmounted before the restore command is issued. After the restore command is completed, the file system can be remounted [13].

3 The snapPVFS

In this section, we describe our proposed scheme, snapPVFS, which is a modified PVFS equipped with the snapshot function. Because of the importance of maintaining the original performance of this parallel file system, we decided to modify the data structure of the PVFS as minimally as possible; hence, the idea of using both the PVFS and ext3cow. Because the PVFS runs on top of an existing file system, such as ext2 or ext3, and because ext3cow is based on ext3, we thought it would be possible to combine the PVFS and ext3cow without modifying the original data structure of the PVFS.

As mentioned in the previous section, there are various kinds of versioning file systems and each system has a different approach. While some file systems can provide a

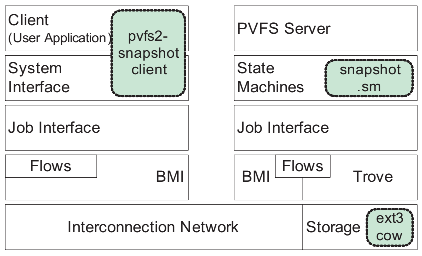

Figure 1. Structure of the snapPVFS

snapshot of an entire file system, other systems can handle snapshots of individual files. Our focus is on the former. In the following sections, we present the structure of our proposed scheme and describe how the PVFS was retrofitted to support the snapshot function.

3.1 System overview

Because we tried to preserve the structure and interfaces of the PVFS, we focused on adding new snapshot components and changing the PVFS as minimally as possible. Furthermore, these added components were designed and implemented in accordance with PVFS development guidelines [14]. As shown in Figure 1, snapPVFS has three new parts: the client-side snapshot command, called ***pvfs2-snapshot***; the server-side state machine, called ***snapshot.sm***; and a modified ext3cow file system on snapPVFS servers. More detailed explanations are given in the following sections. Although there are supplementary codes for the message control, signal definition, request scheduler and so on, we limited our explanations to the major conceptual components of the snapPVFS.

3.2 The global time for an inode epoch counter

The inode epoch counter is an important component of ext3cow. The essence of generating a snapshot is to set the time variable, namely the ***inode epoch counter***, with the new system time. When a file is created or updated after a snapshot has been generated, the time information in the ***inode epoch counter*** is written in the inode of the file. Because ext3cow was developed for a single machine environment, it has only one system time. Using the system time is as straightforward as setting the inode epoch counter. However, in the case of parallel or distributed file systems, there are multiple systems and each of them can have a different time. Owing to this change of environment, we designed the snapPVFS to get the global time before taking a snapshot; furthermore, each snapPVFS server generates a snapshot with the global time. The process of deciding the global time, which is shown in Figure 2, is as follows:

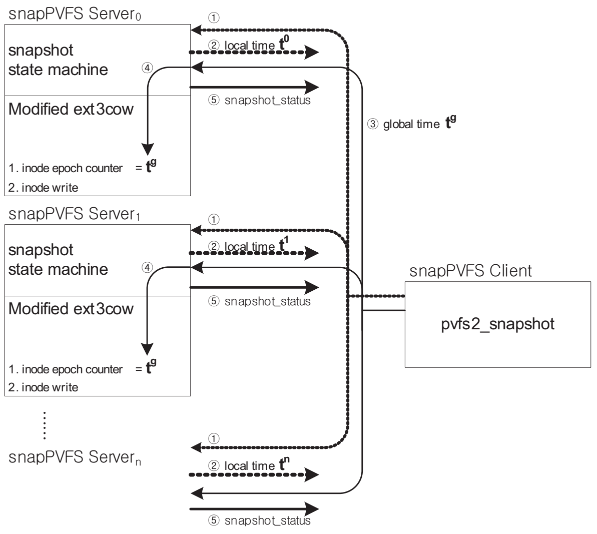

Figure 2. Global time model of the snapPVFS

1. From a snapPVFS client node, a user with root authority executes a ***pvfs2-snapshot*** command.
2. Our snapshot state machine in the snapPVFS servers fetches its own local time and delivers it to ***pvfs2-snapshot***.
3. Each local time is compared by ***pvfs2-snapshot***, which selects the greatest value as the global time and sends it to all the snapPVFS servers.
4. The snapshot state machine passes on the global time to ext3cow. Then ext3cow uses the global time to set the inode epoch counter. Because the original ext3cow uses the system time implicitly, it should be changed to accept the time parameter from the snapshot state machine.
5. The snapshot state machines send the snapshot status to ***pvfs2-snapshot***.

Before the inode epoch counter is changed, the snapPVFS servers are examined to see whether they have ongoing I/O jobs. If there are write or update I/O jobs in progress, the snapshot should be postponed to ensure that the operation is consistent. Thus, in steps 2 and 5, supplementary information must be delivered to pvfs2-snapshot to indicate whether a snapshot can be generated. A detailed explanation of the concept of consistency of operation is given in the next section.

Although the PVFS consists of multiple servers, there is no communication among the servers because of the preservation of the scalability of the parallel file system. Because

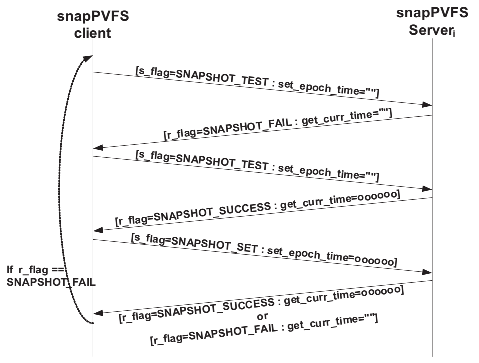
snapPVFS client
snapPVFS Server$_i$
[s_flag=SNAPSHOT_TEST : set_epoch_time=""]
[r_flag=SNAPSHOT_FAIL : get_curr_time=""]
[s_flag=SNAPSHOT_TEST : set_epoch_time=""]
[r_flag=SNAPSHOT_SUCCESS : get_curr_time=oooooo]
[s_flag=SNAPSHOT_SET : set_epoch_time=oooooo]
If r_flag == SNAPSHOT_FAIL
[r_flag=SNAPSHOT_SUCCESS : get_curr_time=oooooo]
or
[r_flag=SNAPSHOT_FAIL : get_curr_time=""]

Figure 3. Message exchange between the snapPVFS server and the client

we also follow the PVFS design policy, we chose not to implement any communication among the snapPVFS servers. Hence, it is impossible for a snapPVFS server to check the status of another server's snapshot, though pvfs2-snapshot in the snapPVFS client can collect information on the status of a snapshot. Therefore, pvfs2-snapshot is in charge of choosing the global time and deciding whether snapshot should be retried.

The snapshot procedure of the snapPVFS in Figure 2 can be divided in two phases. The first phase of pvfs2-snapshot occurs when pvfs2-snapshot checks the status of the snapPVFS servers to determine whether a snapshot is possible; this phase corresponds to steps 1 and 2 in Figure 2. The first phase is periodically iterated until all the snapPVFS servers confirm the suitability of taking a snapshot.

If all the snapPVFS servers confirm the suitability of taking a snapshot, pvfs2-snapshot sends the global time and a message for generating the snapshot to all the snapPVFS servers. This action marks the beginning of the second phase. However, due to the characteristics of distributed systems, when the message is delivered, some snapPVFS servers may already have another I/O job, which can violate the consistency of the snapshot. In this case, the snapPVFS servers send a failure message to pvfs2-snapshot, which then retries the snapshot operation from the first phase, regardless of the results of other snapPVFS servers. Figure 3 shows an example of the message exchange between the snapPVFS server and client.

3.3 Consistent snapshot

As with other file systems, our snapPVFS also prepares the mechanism to ensure consistency in the snapshot. Because the snapPVFS has multiple servers, the exclusive use of ext3cow without a reasonable management scheme may unexpectedly cause an inconsistent state in the file system when a user performs time-shift operations. Additional functionality should therefore be considered as a means of guaranteeing consistency of operation. To follow the PVFS design policy, we also assumed there was no control communication among the snapPVFS servers. Each snapPVFS server tests whether its snapshot is possible and pvfs2-snapshot checks on the success of the snapshots of all the snapPVFS servers.

In our design, the snapPVFS first checks if there is a file write or update request in the snapPVFS servers before issuing a snapshot. Although previous works have reported considerable snapshot granularity, we believe that the snapPVFS supports file-level consistency. To detect whether a snapPVFS server has any I/O jobs, we changed the scheduler in the snapPVFS servers. Each PVFS server has a request scheduler that manages the queue for all I/O operations. Most of these I/O operations are executed concurrently by means of an asynchronous I/O, and the scheduler only manages the number of asynchronous I/O jobs. In contrast, some management requests are served sequentially and immediately. In the current version of the PVFS, only one thread is in charge of executing the main PVFS loop, including the scheduler and the state machines, for these management requests. Because our snapshot state machine is also executed by the main thread, no immediate jobs can interfere with the snapshot state machine. Hence, we fetched the number of asynchronous I/O jobs in the server and used that number as one of the identifiers to test whether a snapPVFS server can take a snapshot or not.

However, when PVFS I/O servers handle the client's I/O request, they manage only a split portion of the original file, which is named ***distribution***, and they don't know the relations between each ***distribution***, even if each ***distribution*** makes up the same file. The major problem of supporting file-level consistency for the snapPVFS is how the snapPVFS client notifies the snapPVFS servers about whether the current distribution is the end of the I/O jobs. Considering the characteristics of the PVFS, we designed two snapPVFSs with a different approach for the purpose of achieving a consistent snapshot.

The first version of our proposed scheme, snapPVFSα, uses a simple method. When a snapshot request arrives at a snapPVFS server, the snapshot state machine checks for the presence of a write-related distribution in the I/O queue. Because there is a time gap between any two distributions, even though the two distributions come from the same I/O job, snapshots can be generated on the way of handling the I/O job. To prevent this situation, snapPVFSα uses another parameter, ***deferred time***, and it is inputted by the administrator. Whenever a snapPVFS server checks its I/O queue and the time gap is less than the given deferred time value, snapPVFSα regards the two distributions as related and postpones the taking of the snapshot. Although this approach is not elaborate, snapPVFSα has the advan-

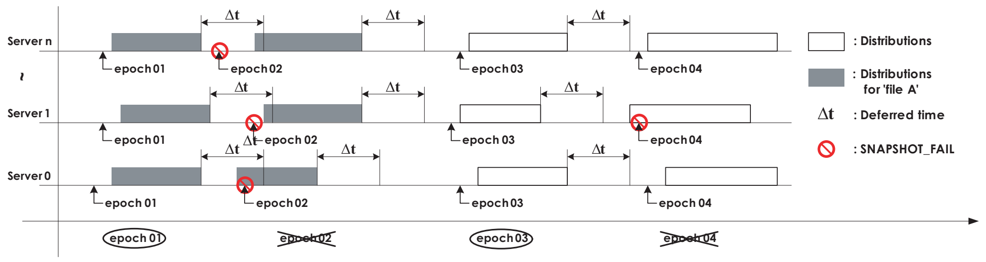

Figure 4. Example of correct snapshots in snapPVFSα

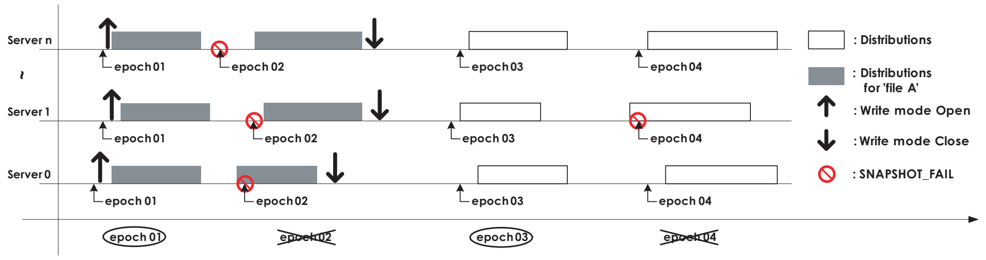

Figure 5. Example of correct snapshots in snapPVFSβ

tage of only having to modify the PVFS server and not the client. Figure 4 shows the basic concept of snapPVFSα.

When PVFS servers manage I/O requests, they handle distributions and don't receive any information as to whether a file is opened or closed. In terms of the second version of the snapPVFS, we modified the way snapPVFS clients send information to the snapPVFS servers about a file being opened or closed with a write mode. With this additional information, snapPVFSβ can find out how many files are opened for a write function. Of course, because of the possibility that only a single independent distribution is located in the I/O queue, the I/O queue must still be checked as in the first version of snapPVFS, though deferred time is no longer used. Figure 5 shows a brief outline of snapPVFSβ.

In the case of ext3cow, taking a snapshot means setting a new inode epoch counter that is used by subsequent I/O operations; the data itself is not changed. Thus, although some servers fail to take snapshots, the fact that the problematic inode epoch counter is not selected for time shifting causes no problems. Whenever pvfs2-snapshot gets the message SNAPSHOT_FAIL for the second phase, it discards the epoch time and retries the snapshot operations from the first step. As shown in Figures 4 and 5, a user can get only the correct epoch time, such as ***epoch 01*** and ***epoch 03***.

3.4 Snapshot state machine and commands

A snapshot state machine is located in every snapPVFS server. When a snapshot request is delivered from pvfs2-snapshot, it generates messages in the sequence shown in Figure 3. Figures 6 and 7 describe the pseudocode of the snapshot state machine and pvfs2-snapshot. Figure 8 also shows an example of when the pvfs2-snapshot command is executed. In the example, because the snapPVFS servers were serving another client's write request, the snapshot was delayed for a while.

```
receive message from pvfs2-snapshot
if(s_flag == SNAPSHOT_TEST)                  /*   PHASE #1   */
   get_curr_time <- system time
   if(snapshot is possilbe)
      r_flag <- SNAPSHOT_FAIL
   else
      r_flag <- SNAPSHOT_SUCCESS
else                                         /*   PHASE #2   */
   if(snapshot is possilbe)
      ioctl(fd, EXT3COW_IOC_TAKESNAPSHOT2, set_epoch_time)
      r_flag <- SNAPSHOT_SUCCESS
   else
      r_flag <- SNAPSHOT_FAIL
send message to pvfs2-snapshot
```

Figure 6. Pseudocode of the snapshot state machine (a snapPVFS server)

As mentioned, because of our focus on taking snapshots of an entire file system, the time-shift interfaces for each individual file are not prepared. Instead of making time-shift interfaces, we altered the startup process of the PVFS server so that it can receive the additional parameter of the epoch time. Figure 9 shows an example of starting snapPVFS in time-shift mode. Because the snapPVFS server was executed with the epoch time, the server was launched with a time-shift mode.

When executed without a description of the epoch time, the snapPVFS is launched in the normal mode. If the ad-

```
get snapPVFS server list
while(1){
   while(1){                              /*   PHASE #1   */
      s_flag <- SNAPSHOT_TEST
      send message to snapshot.sm in all servers
      receive message from snapshot.sm in all servers
      if(ALL SNAPSHOT_SUCCESS)
         define global_time
         break
      else
         wait
   }
   s_flag <- SNAPSHOT_SET                 /*   PHASE #2   */
   set_epoch_time <- global_time
   send message to snapshot.sm in all servers
   receive message from snapshot.sm in all servers
   if(ALL SNAPSHOT_SUCCESS)
      break
   else
      wait
}
```

Figure 7. Pseudocode of pvfs2-snapshot (a snapPVFS client)

```
[root@c0-14]# pvfs2-snapshot -m /mnt/pvfs2 -i 500 -p
[Test - Retry]
[Test - Retry]
Meta server
----------------------------------------------------------
server: tcp://c0-0:3334 |  current server time : 1201854818

I/O server
----------------------------------------------------------
server: tcp://c0-1:3334 |  current server time : 1201855099
server: tcp://c0-2:3334 |  current server time : 1201855099
server: tcp://c0-3:3334 |  current server time : 1201854726
server: tcp://c0-4:3334 |  current server time : 1201854648
----------------------------------------------------------
Snapshot OK!       Epoch_time = 1201855099
Elapsed time: 1103889 (usec)
Retry Count: 2
[root@c0-14]#
```

Figure 8. Example of a snapshot command

ministrator inputs the correct epoch time, the snapPVFS retrieves and serves an old version of the file system in a read-only mode. For clients, the snapPVFS must be remounted so that the clients can get the time-shifted data. If the snapPVFS is in the time-shifted mode, the I/O operations for the write or update functions are all prohibited.

4 Performance evaluation

We implemented two snapPVFSs on a Linux cluster system. Table 1 shows the hardware components of our testbed and software environments on which our snapPVFS were implemented. Our testbed consists of one metadata server, four IO servers and some client nodes. We verified that the snapPVFS generates snapshots properly and its old version of data can be retrieved correctly. However, because our snapPVFS was derived from a parallel file system, we thought it important to sustain the performance of the parallel file system. In this section, we explain the performance and overhead of the snapPVFS.

CPU	AMD Opteron 240	OS	Linux-2.6.20
Memory	2GB	PVFS	PVFS-2.6.3 (pvfs-lkv21.patch)
HDD	SCSI 35GB $\times$ 2	Versioning File System	linux-2.6.20.3-ext3cow.patch
Network	Gigabit Ethernet	MPI	mpich2-1.0.6

Table 1. Hardware and software environments

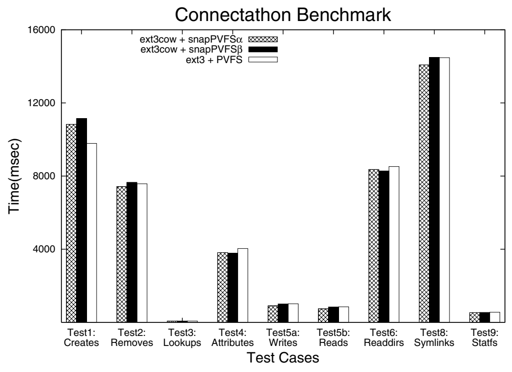

Figure 10. Results from the Connectathon benchmark suite

4.1 Connectathon NFS test suite

The Connectathon NFS Test Suite checks the correctness and performance of I/O operations[15]. Figure 10 shows the results from the Connectathon test. Because the current version of the PVFS doesn't support hard link operation, it was impossible to run the seventh subtest, which includes a hard link system call. Compared with a general file system, the PVFS has a long execution time; furthermore, the performance of the snapPVFS is similar to that of the PVFS. However, in this test, we focused on verifying the correctness of the file system. The performance issues are explained in the following sections.

4.2 Bonnie

The current version of the PVFS can be accessed via two low-level I/O interfaces, the UNIX API and the MPI-IO.

```
[root@c0-0 etc]# pvfs2-server ./pvfs2-fs.conf ./pvfs2-server.conf-c0-0 -t 1201855099
[D 17:39:33.705810] PVFS2 Server version 2.6.3 starting.
[E 02/01 17:39] Storage Path = /data01/pvfs2-storage-space@1201855099
[root@c0-0 etc]# ps ax | grep pvfs2
 6250 ?        Ssl    0:00 pvfs2-server ./pvfs2-fs.conf ./pvfs2-server.conf-c0-0 -t 1201855099
 6264 pts/0    S+     0:00 grep pvfs2
```

Figure 9. Example of the snapPVFS while running with a time shift

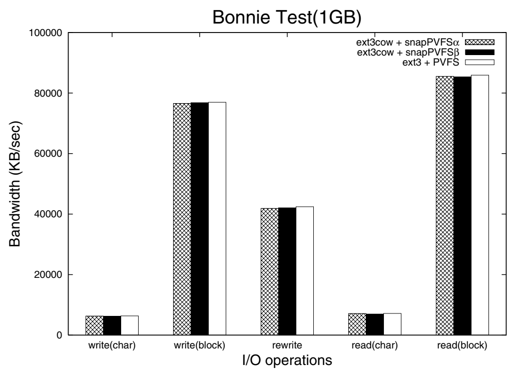

Figure 11. Results from the Bonnie test

The purpose of using a Bonnie test, the famous benchmark for Unix-based file systems[16], is to measure the performance of the snapPVFS when it is accessed through the UNIX API. As shown in Figure 11, the benchmark results indicate that the snapPVFS and the original PVFS are comparable. In practice, the performance degradation of the snapPVFS is in the range of about 0.5% to 0.6%, especially in the case of the block-write and block-read experiments.

We also measured the overhead of taking snapshots. While writing and reading a 1 GB file, we periodically called the snapshot command. As mentioned, when a file is being generated, the snapshot is postponed, which causes an overhead in terms of repeated efforts to take the snapshot. All the snapshots are generated while a file is being read. As shown in Figure 12, while the overhead of repeated efforts to take a snapshot is about 1.8%, the overhead of generating snapshots is about 8.77%. To generate several snapshots for the benchmark, we frequently called the snapshot command. This is the likely reason the performance degradation is independent of the number of snapshots. When a snapshot is generated, the super block of the snapPVFS storage space is updated. The updating process is the main overhead of taking the first snapshot, though subsequent snapshot commands simply refer to the unified buffers in the snapPVFS servers.

4.3 IOR

IOR is a parallel file system benchmark based on the MPI[17]. In this and following sections, we describe the performance of the snapPVFS with the MPI-IO. Because we designed snapPVFSβ so that the snapPVFS client transfers information to the snapPVFS server regarding file open and close functions, the MPI-IO library in mpich2[18] was also changed to support these transfers.

Four process nodes participated in the IOR benchmark as clients and they generated a 2 GB file. In the case of a single I/O benchmark, such as Bonnie, a client accesses the snapPVFS servers almost in sequence. Therefore, even

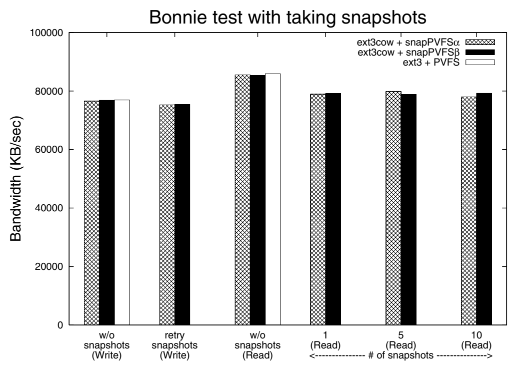

Figure 12. Results from the Bonnie test (block-write and block-read) for the taking of snapshots

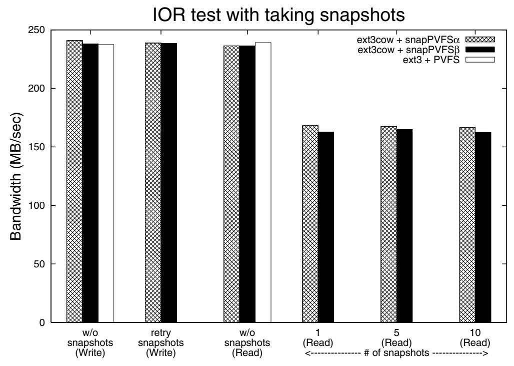

Figure 13. Results from the IOR test for the taking of snapshots

if a snapshot request is delivered to the snapPVFS servers during the single I/O benchmark, some of the snapPVFS servers are likely to be in an idle state. However, IOR can enable the snapPVFS to operate at full capacity and, when the snapPVFS servers get a snapshot request, almost all of them are in a busy state. We therefore expected more serious performance degradation in the IOR test with regard to the taking of snapshots. Figure 13 shows the results of the IOR benchmarks. As in the Bonnie test, the overhead of repeated attempts to take a snapshot is about 1.2%. Accordingly, as expected, the overhead of generating snapshots increases to about 31.26%.

4.4 BTIO

BTIO is a parallel I/O benchmark based on the block-tridiagonal problem of the NAS parallel benchmarks[19]. Unlike Bonnie and IOR, BTIO combines computations and I/O operations, and we think BTIO is more suitable for representing HPC applications. Owing to the characteristics

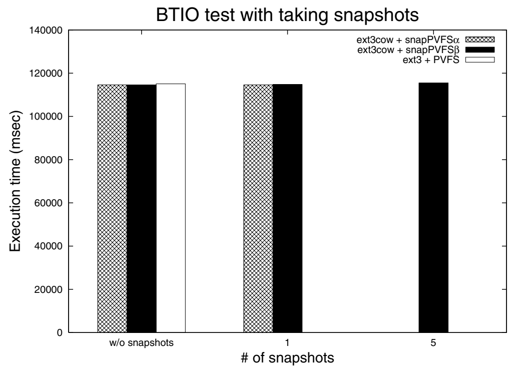

Figure 14. Results from the BTIO (A Class, 4 clients) test for the taking of snapshots

of BTIO, the performance difference among the results is negligible, as shown in Figure 14.

Figure 14 also shows the results of the different behavior of two snapPVFSs. BTIO alternately performs I/O operations and computations. Hence, for the consistency of the snapshot, the deferred time of snapPVFSα must be longer than each computation time. One of the disadvantages of snapPVFSα is that even though a write function is completed the snapshot must be delayed until the deferred time has elapsed. Accordingly, snapPVFSα failed to generate five snapshots before the BTIO operation was completed[1]. On the other hand, because snapPVFSβ can generate a snapshot immediately upon the closure of a file opened for a write function, five snapshots were generated on time.

5 Conclusion

We have considered the possibility of taking a snapshot of an entire file system for the PVFS. To guarantee a consistent snapshot, we implemented two snapPVFSs. We verified that our implementation works correctly as expected. To measure the overhead of the snapPVFS in diverse situations, we tested it with various benchmarks. The overhead was generally reasonable but, in the case of I/O-intensive parallel benchmarks, the process of taking a snapshot caused considerable performance degradation. However, if snapshots are managed exclusively by administrators, this unwished situation can be avoided.

Although the snapPVFS guarantees consistent and atomic operation, if there are any consecutive heavy I/O requests, the snapshot is always postponed. For our future research, we intend to incorporate a quiesce I/O mechanism into a new snapPVFS to overcome this limitation.

[1]When the problem size is increased, because the computation time and the deferred time are also extended, experiments show the same behavior.

References

[1] Parallel Virtual File System, Retrieved June 19, 2008, from http://www.pvfs.org

[2] Chung-hsing Hsu, and Wu-chun Feng, “A Power-Aware Run-Time System for High-Performance Computing,” Proc. of the 2005 ACM/IEEE conference on Supercomputing, pp.1∼1, 2005.

[3] Sudharshan Vazhkudai, and Xiaosong Ma, “Recovering transient data: automated on-demand data reconstruction and offloading for supercomputers,” ACM SIGOPS Operating Systems Review, Vol. 41(1), pp. 14∼18, 2007.

[4] Joseph Tucek, Paul Stanton, Elizabeth Haubert, Ragib Hasan, Larry Brumbaugh, and William Yurcik, “Trade-offs in protecting storage: a meta-data comparison of cryptographic, backup/versioning, immutable/tamper-proof, and redundant storage solutions,” Proc. of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies, pp. 329∼340, 2005.

[5] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew P. Himmer, and Erez Zadok, “A Versatile and User-Oriented Versioning File System,” Proc. of the 3rd USENIX Conference on File and Storage Technologies, pp.115∼128, 2004.

[6] Philip Carns, Walter Ligon III, Robert Ross, and Pete Wyckoff, “BMI: a network abstraction layer for parallel I/O,” Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium, pp. 8∼8, 2005.

[7] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir, “Deciding when to forget in the Elephant file system,” Proc. of the 17th ACM Symposium on Operating Systems Principles, pp.110∼123, 1999.

[8] Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante, “Wayback: A User-level Versioning File System for Linux,” Proc. of the USENIX Annual Technical Conference, pp. 19∼28, 2004.

[9] Zachary Peterson, and Randal Burns, “Ext3cow: A Time-Shifting File System for Regulatory Compliance,” ACM Transactions on Storage, Vol. 1(2), pp. 190∼212, May, 2005.

[10] Neeta Garimella, “Understanding and exploiting snapshot technology for data protection, Part 1: Snapshot technology overview,” Retrieved June 19, 2008, from http://www-128.ibm.com/developerworks/tivoli/library/t-snaptsm1/index.html

[11] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger, “Metadata Efficiency in Versioning File Systems,” Proc. of the 2nd USENIX Conference on File and Storage Technologies, pp. 43∼58, 2003

[12] “Enterprise File Management with GPFS 3.2,” Retrieved June 19, 2008, from ftp://ftp.software.ibm.com/common/ssi/pm/fy/n/clf03001usen/CLF03001USEN.PDF

[13] “General Parallel File System Advanced Administration Guide Version 3.1,” Retrieved June 19, 2008, from http://publib.boulder.ibm.com/epubs/pdf/bl1adv00.pdf

[14] The Pvfs2-developers Archives, Retrieved June 19, 2008, from http://www.beowulf-underground.org/pipermail/pvfs2-developers

[15] Introduction to the Connectathon NFS Testsuite, Retrieved June 19, 2008, from http://www.connectathon.org/nfstests.html

[16] Bonnie, Retrieved June 19, 2008, from http://www.textuality.com/bonnie/

[17] IOR HPC Benchmark, Retrieved June 19, 2008, from http://sourceforge.net/projects/ior-sio/

[18] MPICH2, Retrieved June 19, 2008, from http://www.mcs.anl.gov/research/projects/mpich2/

[19] NAS Parallel Benchmarks Retrieved June 19, 2008, from http://www.nas.nasa.gov/Resources/Software/npb.html