# rt-Samba: High Performance Real-Time Video File Access through Samba

HIROSHI MINE[†1], TAKU SHIMOSAWA[†1], SOKI SAKURAI[†1], TADASHI TAKEUCHI[†1]

***Abstract***: With online video editing systems, file servers supporting standard file sharing protocol and achieving high I/O performance with real-timeness are required to reduce the overall system cost. AVFS is a file system optimized for the efficient storing and access of video files. Samba is a widely used file server application and enables client terminals to access files via a network. Since Samba is not adapted to efficiently process accesses to video files using features provided by AVFS, modifications of Samba such as the addition of deadline information to I/O requests are required to benefit from AVFS. In this paper, we propose rt-Samba, the Samba-AVFS cooperation method composed of a Samba VFS module and an I/O surrogate daemon using real-time features provided by AVFS. Using this method, I/O access performance is improved while keeping the amount of modifications in Samba low. Furthermore, the standard CIFS protocol is preserved, allowing the use of unmodified editing stations for accessing files. Evaluation results of a Linux file server implementation of our proposed method show that rt-Samba achieves 3 times higher read throughput and 1.1 times higher write throughput keeping the storage I/O stability compared to an original Samba server accessing files on Ext4.

***Keywords***: File system, Network file system, QoS, Media streaming, CIFS, Samba

## 1. Introduction

Online video editing systems enable multiple terminals to edit and preview video files stored in remote file servers via network[14]. Such video production systems avoid the need for large capacity local storage in editing stations. Furthermore, editing work can be performed immediately, without first waiting for potentially long file transfers between editing station and video file archive server[16]. Another benefit of remotely accessing files for video editing is that only a single copy of video files can exist on a file server, thus improving storage efficiency of the overall system[17].

In the editing stations, commodity computers with non-linear video editing software supporting de-facto standard file sharing protocol are practically used to reduce the system cost. Also in the stations, uncompressed video files with very large bit rates are commonly used to prevent deterioration of quality of the videos during editing operations. Hence, with such online video editing systems, besides the real-time aspect of I/O guaranteeing smooth preview, the I/O access performance of the file server is critical[1][15], because it determines the number of editing stations that a single server can simultaneously support, and thus the overall system cost. Therefore, in such video editing systems, file servers supporting standard file sharing protocol and achieving high I/O performance with real-timeness are required.

The Audio/Video File System (AVFS)[2][3] is a file system optimized for the efficient storing and access of video files. Leveraging the features of video files such as large file size and the constraint of playback bit rates, AVFS can achieve high and stable disk I/O performance through the use of large block size and real-time I/O scheduling.

Samba[4] is a widely used file server application. Samba enables client terminals to access files stored in the remote file servers using Common Internet File System (CIFS)[13] protocol via network[18]. Samba does not rely on any particular features of the local file system being used, and thus is not adapted to efficiently process accesses to video files using features provided by AVFS, such as real-time I/O scheduling. Therefore, modifications of Samba such as the addition of deadline information to I/O requests are required to benefit from AVFS.

In this paper, we present rt-Samba, a Samba-AVFS cooperation method composed of an extension module for Samba and an I/O surrogate daemon which controls file I/O to AVFS on behalf of Samba. With this method, I/O access performance and its real-time properties are improved while keeping the amount of modifications in Samba low and preserving the standard CIFS protocol used by editing stations for accessing files.

The remainder of this paper is organized as follows. Section 2 describes video file optimized features of AVFS and discusses problems to make Samba exploit AVFS I/O access performance. We propose rt-Samba, a Samba-AVFS cooperation method and describe its implementation in details in Section 3. Section 4 presents and discusses experiments results. Related work is discussed in Section 5 and concluding remarks are given in Section 6.

## 2. Background

AVFS is a file system developed in Hitachi optimized for the efficient storing and access to video files. AVFS is composed of a file system and a real-time I/O scheduler cooperating to provide efficient processing of real-time disk accesses with quality of service (QoS) guaranteed. While preserving the standard POSIX set of system calls for best effort I/O operations, AVFS provides a set of APIs implemented with ioctl() system call allowing applications to specify deadlines for real-time requests, thus implementing traffic differentiation. AVFS also implements a block allocation policy resulting in stable performance, independent of the files accessed. The disk scheduler is

†1 Research and Development Group, Hitachi, Ltd.

ⓒ 2016 Information Processing Society of Japan

optimized to deliver high real-time disk throughput with the QoS guarantee while minimizing the disk utilization rate through aggressive seek overhead reduction.

Samba is open source software (OSS) widely used as a file server application. Samba provides network file sharing functions using CIFS protocol which is supported by most modern operating systems. Samba enables client terminals to access files stored in the file systems of file servers via a network as shown in **Figure 1**.

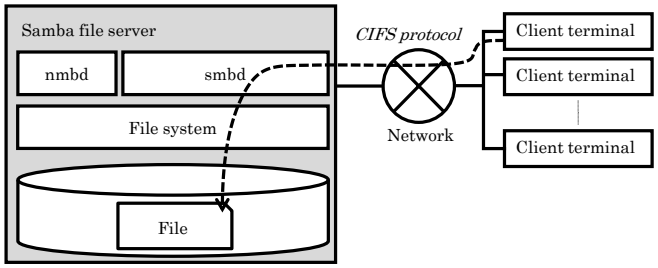


Figure 1 Samba file server.

Samba consists of two daemon processes, nmbd and smbd. The former provides NetBIOS[12] naming service to the clients. The latter provides CIFS file I/O operations to the clients, and thus performs actual I/O access to the files stored in the local file system of the file server. smbd uses virtual file system (VFS) functions to access the files. The VFS functions are implemented as a library using system calls provided by underlying OS. Hence any file system implementing the standard system calls, such as Ext4[8], is used in Samba file server. Samba also defines a module interface to extend its VFS operations. The module implementing the interface is called VFS module. The VFS module is loaded into smbd at run time using the dynamic link mechanism provided by the OS, and then hooks any VFS functions called by smbd.

As mentioned above, to obtain the advantage of AVFS such as the real-time I/O scheduler, application programs are required to use the dedicated APIs to specify the deadlines for real-time requests. Samba is no exception. Therefore, modifications of smbd such as calculating each deadline of I/O request satisfying the constraint of playback bit rate and adding the information to the I/O request by using the dedicated APIs are required to benefit from AVFS.

## 3. rt-Samba

rt-Samba addresses the problems of the standard Samba implementation for real-time file accesses by introducing a new VFS I/O module for Samba and an I/O surrogate daemon which controls file I/O to AVFS.

The I/O module is embedded into Samba as one of its VFS extension modules. The I/O module preempts I/O requests which are issued to local file systems by Samba via the VFS interface. The I/O module adds the path name and the offset of the files to be accessed to these preempted requests, and forwards them to the I/O surrogate daemon.

The I/O surrogate daemon aggregates the I/O requests forwarded from the I/O module and adds control information such as deadline information to the I/O requests on behalf of Samba. The I/O surrogate daemon issues the I/O requests to AVFS as asynchronous I/O requests via its dedicated APIs implemented with an ioctl() system call so that the control information can be added to each I/O request. Here the control information is determined by the configuration information of the I/O surrogate daemon and its dynamic I/O pattern recognition function. When the asynchronous I/O requests completes, the I/O surrogate daemon returns the results of the I/O requests to Samba via the I/O module.

The Samba-AVFS cooperation method makes the I/O module and the I/O surrogate daemon cooperate with each other and improves I/O access performance of Samba keeping the amount of modifications of Samba minimum and the standard CIFS protocol used by editing stations untouched.

### 3.1. Implementation

In this section, we describe the implementation details of our Samba-AVFS cooperation method.

#### 3.1.1. Overview

We implemented the Samba-AVFS cooperation method on a Linux Samba file server. The implementation overview is illustrated in **Figure 2**. In this research, we implemented a VFS AVFS module (vfs_avfs.so) as the I/O module for Samba and an asynchronous I/O daemon (aiod) as the I/O surrogate daemon.

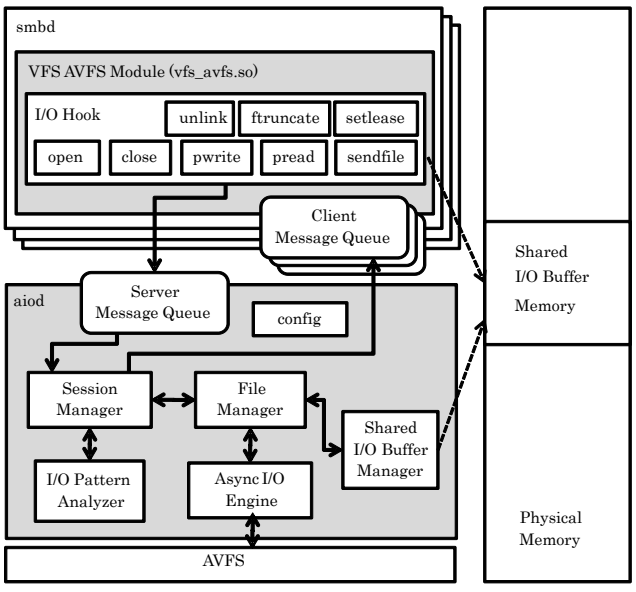


Figure 2 Implementation of Samba-AVFS Cooperation Method.

The VFS AVFS module is embedded in smbd, which is a Samba daemon process. The smbd and aiod processes exchange disk I/O requests and results via POSIX message queues. Note that there would be more than one smbd processes on a file server because smbd forks on accepting a new connection from a client, while there is only one aiod process on the same server. Each smbd and aiod has its own message queue.

smbd and aiod exchange actual I/O data via shared I/O buffer memory allocated in the main memory of the file server.

ⓒ 2016 Information Processing Society of Japan

The memory region is pinned down on the physical memory of the file server in order to avoid swapping out by the OS. aiod also uses the shared I/O buffer memory as cache memory to aggregate write I/O requests and eliminate unnecessary read for efficiency.

By separating the actual data exchanges from the messages as described above, the amount of data copies performed by the OS that stride across the boundary between the user and kernel spaces can be significantly reduced compared with exchanging the whole data with the messages via message queues.

#### 3.1.2. Inter-Process Communication Interface

aiod and smbd are discrete processes running on the same file server. They perform inter-process communication via POSIX message queues to communicate with each other. The messages are processed as a client-server fashion in which aiod acts as a server and smbd does as a client.

The inter-process communication via message queues is performed as follows:

(1) aiod creates a server message queue and each smbd (more precisely, the VFS AVFS module embedded in it) creates a client message queue as one of their initialization steps.
(2) smbd sends a request message to the server message queue of aiod, and then waits for a reply received from its client message queue.
(3) aiod receives the request message by waiting for the server message queue, processes the request, and then sends the result as a reply message to the client message queue of smbd.
(4) smbd receives the reply via the client message queue and processes the result.

The protocol messages used by aiod and smbd for the inter-process communication are shown in **Table 1**. Each request message has a corresponding reply message. The read and write procedures of the Samba-AVFS cooperation method consist of two steps, and each step uses its own message. The procedures are described with the pertaining modules in the next section.

Table 1 aiod protocol messages.

| Message | Description |
|---|---|
| AIOD_MSG_CONNECT | Request to connect to aiod |
| AIOD_MSG_DISCONNECT | Request to disconnect from aiod |
| AIOD_MSG_OPEN | Request to open a file |
| AIOD_MSG_CLOSE | Request to close an opened file |
| AIOD_MSG_READ0 | Request to read data to shared I/O buffer |
| AIOD_MSG_READ1 | Notify completion of copying data form the I/O buffer |
| AIOD_MSG_WRITE0 | Request to allocate shared I/O buffer to write data |
| AIOD_MSG_WRITE1 | Notify completion of copying data to the I/O buffer |
| AIOD_MSG_FTRUNCATE | Request to change the size of a file |
| AIOD_MSG_UNLINK | Request to remove a file |

#### 3.1.3. Modules Detail

In this section, the VFS AVFS module as the I/O module for Samba and aiod as the I/O surrogate daemon are described in detail respectively.

##### 3.1.3.1. I/O Module for Samba

The VFS AVFS module is implemented as one of VFS extension modules for Samba, vfs_avfs.so, which hooks VFS I/O functions called by Samba.

The operations of Samba and the VFS AVFS module are shown as follows:

(1) On accepting a connection request from each unique client, Samba forks a new daemon process smbd, which has the privilege of the user who requests the connection.
(2) The VFS AVFS module is loaded and initialized when the forked smbd first accesses shared resources on AVFS specified in its configuration file smb.conf.
(3) During initialization, the VFS AVFS module creates a client message queue and sends a connection request message which includes the name of the client message queue to the server message queue of aiod whose name is predefined.
(4) The VFS AVFS module obtains the name and size of the shared memory region allocated by aiod for shared I/O buffer as a reply of the connection request from aiod, and maps the region to its own virtual memory space.
(5) The VFS AVFS module removes the client message queue with unlink() immediately after establishing the connection with aiod. The resource used for the client message queue is released when the forked smbd terminates.

The VFS AVFS module achieves the operations above by hooking the VFS functions. The following sections examine the VFS functions hooked and describe how the module works in the functions.

###### 3.1.3.1.1. open

The VFS AVFS module hooks all open() function calls. The module opens a file internally then requests aiod to open the same file. If aiod opened the file successfully, the VFS AVFS module marks the file as the target of further hooking. After that, the VFS function calls described below to the marked files are hooked by the VFS AVFS module. The other VFS function calls such as fstat() are processed by smbd with the file descriptor opened internally.

###### 3.1.3.1.2. close

The VFS AVFS module hooks close() function calls for the marked files, and requests aiod to close the file opened by aiod. Then the VFS AVFS module unmarks the file and closes the file descriptor opened internally.

###### 3.1.3.1.3. pread

The VFS AVFS module hooks pread() function calls for the marked files, and requests aiod to read data from the file on behalf of smbd. First the module sends the file offset and data length to be read to aiod as the READ0 request, and receives the

ⓒ 2016 Information Processing Society of Japan

offset and length of the shared I/O buffer in which the data read by aiod is filled. Then the module copies the data from the specified region of the shared I/O buffer to the memory region specified by the upper layer i.e. smbd. On completing the data copy, the VFS AVFS module sends the READ1 request to the aiod in order to notify aiod of the completion of the use of the share I/O buffer region.

#### 3.1.3.1.4. pwrite

The VFS AVFS module hooks pwrite() function calls for the marked files, and requests aiod to write date to the file on behalf of smbd. First the module sends the file offset and data length to be written to aiod as the WRITE0 request, and receives the offset and length of the shared I/O buffer from aiod as the reply. Then the module copies the data to be written to the file to the specified region of the shared I/O buffer. On completing the data copy, the VFS AVFS module sends the WRITE1 request to the aiod, and receives the length of the data written actually to the file as the reply.

#### 3.1.3.1.5. sendfile

The VFS AVFS module hooks sendfile() function calls for the marked files, and requests aiod to read data and send it to a socket specified by the upper layer. The process of reading data is similar to pread(). The module sends the read data to a socket instead of copying it to a memory region.

#### 3.1.3.1.6. unlink

The VFS AVFS module hooks unlink() function calls, and requests aiod to remove a file. On receiving file removal request, aiod discards the data blocks of the file cached in the shared I/O buffer memory. The file is removed actually by the VFS AVFS module in smbd.

#### 3.1.3.1.7. ftruncate

The VFS AVFS module hooks ftruncate() function calls for the marked files, and requests aiod to change the file size. If the request is to shorten the size, aiod discards the data block of the truncated part of the file cached in the share I/O buffer memory. The file size is changed actually by the VFS AVFS module in smbd.

#### 3.1.3.1.8. setlease

The VFS AVFS module hooks setlease() function calls, and just ignore them. This is because that both smbd and aiod open the same file at the same time but no other process opens the file in the supposed use case of the Samba-AVFS cooperation method.

### 3.1.3.2. I/O Surrogate Daemon

The I/O surrogate daemon aiod runs as a separate process from smbd. aiod performs inter-process communication with the VFS AVFS module embedded in smbd via POSIX message queues, and execute asynchronous disk I/O to AVFS disk volumes on behalf of smbd. The internal modules of which aiod consists and their brief descriptions are shown in **Table 2**. In these modules, the message manager, the session manager, and the asynchronous I/O engine modules have threads which provide execution contexts.

Table 2 aiod internal modules.

| Module | Description |
|---|---|
| Shared I/O buffer manger | Manages shared I/O buffer |
| Message manager | Manages server message queue and process messages |
| Session manager | Manages session with VFS AVFS modules |
| File manager | Manages files |
| Asynchronous I/O engine | Executes asynchronous I/O requests to AVFS disks |
| I/O pattern analyzer | Analyze I/O patterns and determine I/O parameters |
| Config | Processes information specified in a configuration file |

The functions of the internal modules of aiod are described below.

#### 3.1.3.2.1. Shared I/O Buffer Manager

The shared I/O buffer manager module manages I/O buffers allocated on shared memory, which are used by aiod and smbd to exchange actual I/O data which is written to or read from AVFS disks. During the initialization of aiod, the module allocate shared memory region specified in the configuration file with a predefined shared memory name. The shared I/O buffer manager module maps the allocated memory to the virtual memory space of aiod as unswappable, and zeros it out. At this time, actual memory access is performed and all the mapped shared memory is pinned down on the physical memory. The module divides the allocated shared memory into the blocks whose size is specified in the configuration file, and manages them with the used and unused block queues.

#### 3.1.3.2.2. Message Manager

The message manager module manages the server message queue from which aiod receives request messages from smbd, and processes messages. During the initialization of aiod, the module creates a POSIX message queue for the server message queue with a predefined name, and starts a message manager thread which processes sending and receiving of messages. The message manager thread receives messages from the server message queue and does each process requested by the message.

When the message is a connection request, the message manager thread registers a new session with the process id (PID) of smbd, which requested the connection, to the session manager module, and sends the reply to the client message queue.

When the message is a disconnection request, the thread asks the session manager module to discard the registered session. Note that the message manager thread only set the flag of session discard in its context and the session manager thread actually discard the flagged session in its context in order to avoid deadlock.

ⓒ 2016 Information Processing Society of Japan

When the messages are open, close, or removal of a file, the message manager thread does the corresponding file manipulation in the file manager module via the session manager module, and sends the reply to the client message queue.

When the messages are read or write of file data, the message manager thread transfers file I/O requests to the asynchronous I/O engine via the session management module. If the data is already cached on the shared I/O buffer, however, the message manager thread sends the reply without switching the context to the asynchronous I/O engine in order to increase response performance on cache hits.

#### 3.1.3.2.3. Session Manager

The session manager module manages the connections between aiod and smbd as sessions. The module has a thread which judges the validity of sessions and discards invalid sessions. During the initialization of aiod, the session manager creates a session queue which contains session data structures, and starts a session manager thread.

The session data structure contains the PID of smbd and a session file queue which contains session file data structures corresponding to open files. The session file data structure contains references of file data structures which contain information of open files. The session file data also contains a session file I/O queue which contains session file I/O data structures corresponding to file I/O requests under execution. The session between aiod and smbd are associated with the PIDs of smbd. In the same way, the open files are associated with file ids (FIDs) and the I/O requests under execution are with I/O ids (IOIDs) allocated uniquely in aiod.

The session manager thread is invoked periodically and judges the validity of the sessions and discards invalid sessions. The judgment of the validity is made by checking for each session data structures whether the process of the PID exists or not. If the process does not exist, the thread set the session discard flag of the data structure assuming that smbd which had the PID has already terminated without requesting aiod to disconnect. Hereby the validity can be judged even though the connection between aiod and smbd with the message queues is actually connection-less. The thread discards the session by removing the session data structure from the session queue and releasing the resources allocated to the session.

#### 3.1.3.2.4. File Manager

The file manger module manages files which are opened in aiod and file blocks cached on the shared I/O buffer. During the initialization of aiod, the module creates a file queue which contains file data structures.

The file data structure contains a file descriptor of a file opened in aoid, an i-node number of the file, a device number of the file system, a size of the file, and a file block queue which contains file block data structures. The file block data structure contains the references of the cached file blocks held on the shared I/O buffer.

Each file data structure and file block data structure has its own reference counter. The reference of sole file data structure or file block data structure is used to access identical file or file data region. Though the file data structures and the file block data structures whose reference counters are zero could be released at any moment, they are not released instantly and the file data blocks on the shared I/O buffer referenced by the file block data structures are used as cached data. When unused shared I/O buffer runs out, the shared I/O buffer allocated to the file data block of the least recently used file whose reference counter is zero is released and recycled.

#### 3.1.3.2.5. Async I/O Engine

The asynchronous I/O engine module controls the file I/O accesses executed in aiod on behalf of smbd. The engine has a thread which performs actual I/O accesses with the AVFS asynchronous I/O APIs. During the initialization of aiod, the asynchronous I/O engine creates a file I/O queue and an asynchronous I/O queue, and starts an asynchronous I/O engine thread.

The file I/O queue contains file I/O data structures to maintain file I/O requests transferred from the message manager thread. The asynchronous I/O queue contains asynchronous I/O data structure to maintain the AVFS asynchronous I/O requests under execution.

When both the file I/O queue and the asynchronous I/O queue are empty, the asynchronous I/O engine thread sleeps until notified by a signal generated when a file I/O request is enqueued to the file I/O queue. If the request is executable, the thread issues the AVFS asynchronous I/O request with the deadline information calculated by the I/O pattern analyzer via the AVFS asynchronous I/O API, and enqueues the file I/O request as the asynchronous I/O data structure to the asynchronous I/O queue. The unexecuted file I/O requests remain in the file I/O queue. Here the judgment of the excitability of the file I/O request is made by referring the read and write counts and the lock state of the target file block.

The asynchronous I/O engine thread waits for the completion of the AVFS asynchronous I/O requests contained in the asynchronous I/O queue by using the AVFS asynchronous I/O polling API. The wait could be interrupted by a signal generated on arrival of a new file I/O requests.

Returning from the AVFS polling API, the asynchronous I/O engine thread checks the completion of each asynchronous I/O request under execution, and sends the reply to the client message queue if the request is completed. After checking all the asynchronous I/O requests under execution, the thread goes back to the aforementioned check of the both the file I/O queue and the asynchronous I/O queue.

#### 3.1.3.2.6. I/O Pattern Analyzer

The I/O pattern analyzer module recognizes the I/O patterns of the file I/O accesses in aiod, and determines prefetching of file blocks and AVFS asynchronous I/O parameters. The analyzer also calculates deadline information for an AVFS asynchronous I/O parameter for files which match file path patterns and have bitrate information specified in the configuration file. The calculated deadline information is used by the asynchronous I/O engine on issuing an AVFS asynchronous I/O request.

ⓒ 2016 Information Processing Society of Japan

#### 3.1.3.2.7. Config

The config module manages the configuration information of aiod. During initialization of aiod, the config module reads a configuration file and parses keys and values stored in INI style. The module provides references of keys and values on request of the other modules.

### 3.1.4. Implementation Steps

The code steps of the implementation of the Samba-AVFS cooperation modules written in the C language are shown in **Table 3**. Note that the VFS AVFS module is implemented as a VFS extension module for Samba so that no modification of smbd itself is required. Therefore the original binary image of Samba provided with the Linux distribution can be used without any change.

Table 3 Implementation steps of rt-Samba

| Module | | KSteps |
|---|---|---|
| VFS AVFS | | 1.5 |
| aiod | | 5.7 |
| | Shared I/O buffer manager | 0.3 |
| | Message queue manager | 0.5 |
| | Session manager | 1.1 |
| | File manger | 0.7 |
| | Asynchronous I/O engine | 1.0 |
| | I/O pattern analyzer | 0.4 |
| | Config | 0.7 |
| | Others | 1.0 |

## 4. Evaluation

In this section, we evaluate the Samba-AVFS cooperation method by experiment with the implementation prototype and discuss the results.

### 4.1. Experimental Environment

The evaluation of the Samba-AVFS cooperation method is performed by measuring the disk I/O throughput of a Linux Samba file server with and without aiod. **Figure 3** illustrates our experimental environment which simulates a typical online video editing system. We prepare one DELL PowerEdge R710 server for a Linux file server on which Samba and aiod run as file server application software. Ten Hitachi HA8000-110 servers and two HA8000-220 servers are used as client video editing terminals which generate CIFS read and write requests to the Linux file server. These machines are connected each other via a 10Gbps Ethernet switch.

The specifications of the file server are described in **Table 4**. The file server has a storage subsystem which is attached directly to the server via a 4GB FC-AL connection. The storage subsystem has two RAID5 logical units (LUs) each consists of four data disks and one parity disk. One LU on the storage is formatted as an Ext4 file system with mkfs.ext4 default parameters, the other LU is formatted as an AVFS file system whose data block size is 8MB.

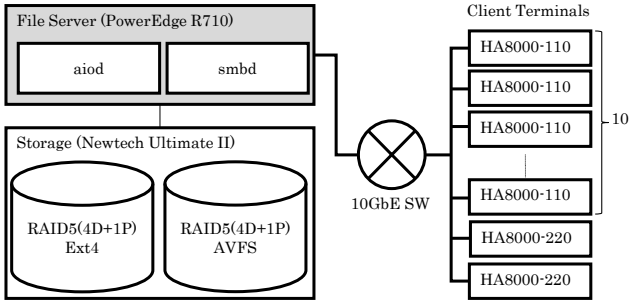


Figure 3 Experimental environment.

Table 4 Linux file server specification.

| Component | Description |
|---|---|
| Manufacturer<br>Model | DELL PowerEdge R710 |
| Processor | Intel® Xeon® E5506@2.13GHz 4core x 2CPUs (total 8 cores) |
| Memory | 24GB DDR3 |
| Host Bus Adapter | 1port 4GB FC-AL Qlogic ISP2432 |
| Storage Subsystem | Newtech Ultimate II, 256MB cache memory<br>SATA 250GB x 12, RAID5(4D+1P) x 2LUs + 2spare drives, 64KB stripe size |
| Server Software | Asianux 3 (Linux-2.6.18-194.1.AXS3)<br>samba.x86_64 3.0.33-3.28.0.1.AXS3 |

Table 5 Storage I/O performance.

| I/O size[KB] | 64 | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
|---|---|---|---|---|---|---|---|
| Random Read[Mbps] | 35.0 | 204.5 | 361.2 | 569.9 | 821.7 | 1032.4 | 1216.6 |
| Random Write[Mbps] | 224.3 | 376.3 | 440.9 | 674.0 | 816.6 | 925.2 | 1123.2 |

In order to make the maximum I/O performance of the storage subsystem known beforehand, both random read and random write throughputs of the LUs are measured by running an I/O load generator program on the file server as a preliminary experiment. The results are shown in **Table 5**.

### 4.2. Experimental Results

To evaluate our proposed method, we performed CIFS network file access tests on the experimental environment for the following three cases; 1)smbd running on the Ext4 LU, 2)smbd on the AVFS LU and 3)smbd cooperate with aiod on the AVFS UL. For each case, the clients generated sequential read or write requests to the video files on the file server. The access bitrates were 100Mbps assuming a high definition (HD) video editorial workload in an editing station. We increased the number of clients concurrently requesting file accesses to the server as long as the file server sustained the requests. The storage disk I/O throughput and the CPU usage were measured by running a load monitor program on the file server during the tests.

The results of the read tests are shown in **Figure 4**. The horizontal axis is for the total number of clients concurrently

ⓒ 2016 Information Processing Society of Japan

accessing the file server. The solid lines show the disk read throughput measured against the left vertical axis, and the broken lines show the CPU usage against the right vertical axis. Without aiod, smbd could not provide enough throughputs so that the maximum sustainable numbers of clients both on Ext4 and AVFS were 3. By cooperating with aiod on AVFS, smbd could serve up to 9 clients without any hassle, while infrequent deadline misses were observed when the number of clients reached 10.

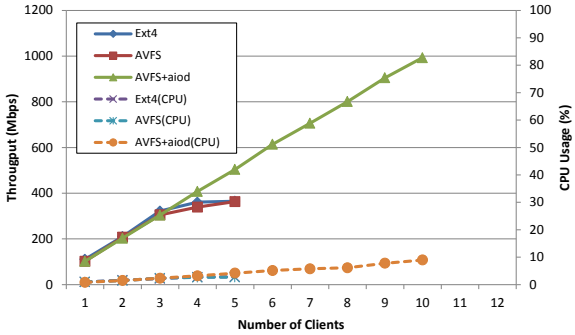


Figure 4 Disk read performance.

In the same fashion of the read, the results of the write tests are show in **Figure 5**. In this case, the maximum sustainable number of clients on AVFS without aiod was 8 and that of on AVFS with aiod was 10. On Ext4 without aiod, smbd got several late disk I/O when serving more than 9 clients.

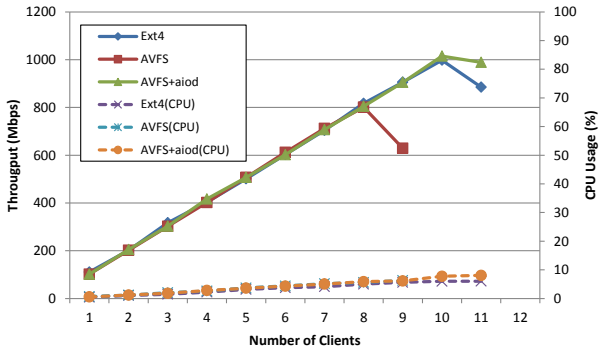


Figure 5 Disk write performance.

### 4.3. Discussion

As shown in Figure 4, smbd with aiod on AVFS achieved 3 times as much disk read throughput as smbd without aiod on Ext4 or AVFS did. The results of smbd without aiod on Ext4 and on AVFS were similar because smbd could not use the dedicated APIs of AVFS and activate the AVFS I/O improvement functions without aiod. The high throughput of smbd with aiod on AVFS was achieved by the large block read ahead feature of aiod. No deadline miss was observed while smbd with aiod on AVFS serving up to 9 clients owing to the real-time I/O scheduler of AVFS, which was activated by adding deadline information to each I/O request by aiod. Since the throughput reached nearly the maximum storage random read throughput shown in Table 5 and both the network bandwidth and the CPU usage were far below their limit, it is suggested that the performance bottleneck be the storage subsystem.

Referring to Figure 5, the performances of smbd without aiod on Ext4 and with aiod on AVFS look similar. However the two can be distinguished by stability which is essential to treat the real-time media. No deadline miss was observed while smbd with aiod on AVFS serving up to 10 clients, while smbd without aiod on Ext4 got several late disk I/O in serving more than 9 clients. This means that smbd with aiod on AVFS achieved 1.1 times as much stable disk write throughput as smbd without aiod on Ext4 did. The stability came from the real-time I/O scheduler of AVFS. Comparing with on AVFS without aiod, smbd on AVFS with aiod achieved higher write throughput because aiod could aggregate I/O requests by using its large shared memory buffer and make a larger I/O request. In contrast to the read, the write performance of smbd on Ext4 without aiod was unexpectedly high. It is conjectured that Ext4 also has some features of I/O request aggregation. Similar to the read, we believe the bottleneck be the storage subsystem.

## 5. Related Work

As a kind of file systems of OSes, several file systems dedicated for storing audio and video files are developed. These file systems leverage the features of video files such as large file size and predetermined playback bitrate to improve the disk I/O access performance. For instance, a technology is developed to enhance the sequentiality of disk access by using the large file size feature of video files[5]. Also technologies are developed to reduce the head seek over head of disks by performing real-time I/O scheduling with control information such as deadline using the constant bitrate feature of video files[6][7].

Many file servers and network storage systems for storing and delivering real-time media files have been studied and some solutions which have relationship with our method are proposed.

RIO[9] is studied as an object server which handles real-time media such as audio and video. RIO achieves low latency access under high disk load by splitting media object accesses between real-time and non-real-time and performing synchronous and asynchronous disk scheduling. However, RIO is not implemented as a file system with standard interfaces which can be used by the conventional file server applications.

Prism[10] is a network storage system for the multi-media files. Prism classifies I/O requests and provides services suitable for each I/O class. In order to get the desired effect of Prism, client applications need to use dedicated APIs to specify service classes of their I/O requests. Thus the conventional file server applications are required to be modified to use the APIs.

Coop-I/O[11] proposes a method to cooperate applications and an OS aiming for less energy consumption. In addition to the standard file interfaces such as open(), close(), read() and write(), Coop-I/O introduces new system calls which adds control information to the I/O requests. By the same token, the conventional file server applications are required to be modified to adapt to the additional system call semantics.

With the aforementioned works, modifications of applications are inevitable to get the advantage of each proposed features and the cost of modifying the conventional file server applications piles up.

ⓒ 2016 Information Processing Society of Japan

## 6. Concluding Remarks

In this paper, we have presented rt-Samba, a Samba-AVFS cooperation method which consists of Samba VFS module vfs_avfs.so and I/O surrogate daemon aiod. The I/O module is embedded into Samba and preempts I/O requests issued by Samba then passes them to the I/O surrogate damon. The I/O surrogate daemon aggregates the I/O requests into a larger I/O request and issues it to AVFS with deadline information via its dedicated API. Using this method, I/O access performance is improved while keeping the amount of modifications in Samba low while preserving the standard CIFS protocol used by editing stations for accessing files.

Evaluation results have shown that Samba cooperating with the I/O surrogate daemon on AVFS achieved 3 times higher read throughput and 1.1 times higher write throughput than Samba on Ext4. Our method make Samba exploit AVFS I/O access performance to the extent of the storage subsystem bottleneck.

Future work will concentrate on characterizing more precisely the impact of the I/O pattern analyzer and adopting advanced features of the latest CIFS protocol.

## Reference

[1] Mokbel, M.F.; Aref, W.G.; Elbassioni, K.; Kamel, I., "Scalable multimedia disk scheduling", Proceedings. 20th International Conference on Data Engineering, 2004, pp. 498-509, 30 March-2 Apr. 2004

[2] Damien Le Moal, Donald Molaro, and Jorge Campello, "A Real-Time File System for Constrained Quality of Service Applications", In Information Processing Society of Japan, Transactions on Advanced Computing Systems (ACS), vol. 3, no. 1, pp. 61-76, Mar. 2010

[3] Le Moal, D.; Molaro, D.; Bandic, Z.Z., "Stable disk performance with non-sequential data block placement", IEEE 14th International Symposium on Consumer Electronics (ISCE), pp.1-6, 7-10 Jun. 2010

[4] French, M. S., “A New Network File System is Born: Comparison of SMB2, CIFS and NFS”, Proceedings of the Linux Symposium, Volume One, 131-140, Jun. 2007

[5] An-I Andy Wang, Geoff Kuenning, Peter Reiher, and Gerald Popek., “The Conquest file system: Better performance through a disk/persistent-RAM hybrid design”, Trans. Storage, vol.2, no.3, pp.309-348, Aug. 2006

[6] A. L. N. Reddy, Jim Wyllie, and K. B. R. Wijayaratne, “Disk scheduling in a multimedia I/O system”, ACM Trans. Multimedia Comput. Commun. Appl., vol.1, no.1, pp.37-59, Feb. 2005

[7] Hong Li; Cumpson, S.R.; Jochemsen, R.; Korst, J.; Lambert, N., "A scalable HDD video recording solution using a real-time file system," IEEE Transactions on Consumer Electronics, vol.49, no.3, pp. 663- 669, Aug. 2003

[8] A. Mathur, M. Cao, and S. Bhattacharya, “The new EXT4 filesystem: current status and future plans”, in Proc. the 2007 Ottawa Linux Symposium, Ottawa, Canada, Jun. 2007, pp. 21–34.

[9] Richard Muntz, Jose Renato Santos, and Steve Berson, “RIO: a real-time multimedia object server”, SIGMETRICS Perform. Eval. Rev. vol. 25, no. 2, pp.29-35, Sep. 1997

[10] Ravi Wijayaratne and A. L. N. Reddy, “System support for providing integrated services from networked multimedia storage servers”, In Proceedings of the ninth ACM international conference on Multimedia (MULTIMEDIA '01), Oct. 2001

[11] Andreas Weissel, Björn Beutel, and Frank Bellosa, “Cooperative I/O: a novel I/O semantics for energy-aware applications”, In Proceedings of the 5th symposium on Operating systems design and implementation (OSDI '02), Dec. 2002

[12] A. Aggarwal and K. Auerbach. Protocol standard for a netbios service on a tcp/udp transport. IETF Network Working Group RFC 1001, March 1987.

[13] P. J. Leach and D. C. Naik. A common internet file system (cifs/1.0) protocol. IETF Network Working Group RFC Draft, March 1997.

[14] Sumiyoshi, H.; Mochizuki, Y.; Suzuki, S.; Ito, Y.; Orihara, Y.; Yagi, N.; Nakamura, M.; Shimoda, S., "Network-based cooperative TV program production system," Broadcasting, IEEE Transactions on , vol.42, no.3, pp.229,236, Sep 1996

[15] Nou, R.; Giralt, J.; Cortes, T., "Automatic I/O Scheduler Selection through Online Workload Analysis," Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC/ATC), 2012 9th International Conference on , vol., no., pp.431,438, 4-7 Sept. 2012

[16] Owen, S.J., "The changing shape of on-line disc-based editing," Broadcasting Convention, International (Conf. Publ. No. 428) , vol., no., pp.73,78, 12-16 Sep 1996

[17] Bucci, G.; Detti, R.; Pasqui, V.; Nativi, S., "Sharing Multimedia Data Over a Client-Server Network," MultiMedia, IEEE , vol.1, no.3, pp.44,, Autumn/Fall 1994

[18] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. 2008. Measurement and analysis of large-scale network file system workloads. In USENIX 2008 Annual Technical Conference on Annual Technical Conference (ATC'08). USENIX Association, Berkeley, CA, USA, 213-226.

## Acknowledgments

We would like to thank Atsushi Hashimoto, Minoru Urushima and Kenji Hirose of Hitachi Industry & Control Solutions, Ltd. for their support for this work.

ⓒ 2016 Information Processing Society of Japan